Unity User Manual (2018.3 beta)

Leave feedback

Use the Unity Editor to create 2D and 3D games, apps and experiences. Download the Editor at unity3d.com.

The Unity User Manual helps you learn how to use the Unity Editor and its associated services. You can read it from start to finish, or use it as a reference.

If it's your first time using Unity, take a look at the introductory documentation on Working with Unity, and see the Unity Tutorials.

New

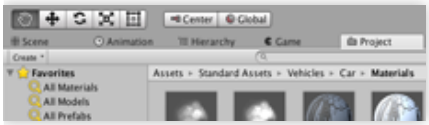
Features introduced in 2018.3: What's New in 2018.3
Upgrading Unity projects from older versions of Unity: Upgrade Guide

Best practice and expert guides

- Best practices from Unity Support engineers: Best Practice Guides
- Expert guides from Unity developers, in their own words: Expert Guides

---

Unity User Manual sections

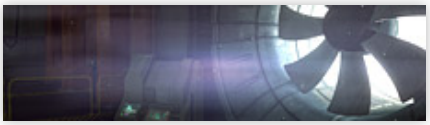


Working in Unity

A complete introduction to the Unity Editor.

Unity 2D

All of the Unity Editor's 2D-specific features including gameplay, **sprites** and physics.



Graphics

The visual aspects of the Unity Editor including **cameras** and lighting.

Physics

Simulation of 3D motion, mass, gravity and **collisions**.

Networking

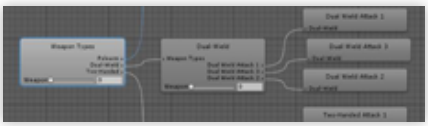
How to implement Multiplayer and networking.

Scripting

Programming your games by using scripting in the Unity Editor.

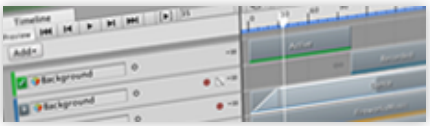
Audio

Audio in the Unity Editor, including clips, sources, listeners, importing and sound settings.



Animation

Animation in the Unity Editor.



Timeline

Cinematics in the Unity Editor, including cut-scenes and game-play sequences.

UI

The Unity Editor's UI system.

Navigation

Navigation in the Unity Editor, including AI and pathfinding.

Unity services

Virtual reality

Contributing to Unity

Suggest modifications to some of the Unity Editor's source code.

Platform specific

Specific information for the many non-desktop and web platforms you can make projects for with the Unity Editor.

Legacy topics

Useful if you are maintaining legacy projects.

Further sources of information

Unity Answers or Unity Forums - here you can ask questions and search answers.

The Unity Knowledge Base - a collection of answers to questions posed to Unity's Support teams.

Tutorials - step by step video and written guides to using the Unity Editor.

Unity Ads Knowledge Base - a guide to including ads in your game.

Everyplay documentation - a guide to the Everyplay mobile game replay platform.

Asset Store help - help on **Asset Store** content sharing.

Known issues

Is a feature not working as you expect it to? It might be an existing Known Issue. Please check using the **Issue Tracker** at issuetracker.unity3d.com.

Documentation versions

Leave feedback

The Unity documentation is the Unity User Manual and Unity Scripting API Reference.

As we continually improve Unity (that is; the Editor and the engine), we add new features, improve existing features, and sometimes remove old features.

With each Unity release, the Unity documentation changes to reflect this, so make sure you use the correct version of the User Manual and Scripting API Reference to match the version of Unity you're using.

The version of the documentation which accompanies the current full release of Unity is always available online at **docs.unity3d.com** .

Installer (offline) documentation

As well as using the online documentation, you can install it locally on your computer when you install the Unity software. Prior to Unity 5.3, the documentation was *always* installed along with the software. From version 5.3 onwards, you can choose whether to install the documentation locally via the Unity Download Assistant.

Documentation updates

We republish the online documentation for the current full release version of Unity regularly with corrections and missing content regularly. (You can see the publish version of the documentation at the bottom of every page.)

If there is a current Unity beta release, we also update the current public beta release documentation with corrections, missing content, and feature updates regularly.

However, note that we do not update and re-publish:

the installer (offline) documentation.

legacy documentation, that is; documentation for Unity versions which are no longer a current beta or full release of Unity.

If you need the latest publish of the documentation offline, you can download it outside of the installer. See the Offline Documentation page for details.

Which documentation version to use

Most people use the latest full release version of Unity and so use the documentation at docs.unity3d.com.

Some people need to use older versions of Unity. This might be the case for you if you are maintaining or in long-term development of a project which uses an older version of Unity.

If you are using an older version of Unity, the installer (offline) documentation matches that version of Unity. However if you chose not to install local documentation, or want to ensure you can access the latest publish of the documentation, you can access older versions of the Unity documentation online. (See list below.)

Legacy documentation

Documentation for Unity versions which are no longer the current full release or beta release version of Unity.

This documentation is frozen - we do not correct, update or republish it. However Long-term support versions may be supported for a longer time.

Unity 2017 documentation

Version 2017.4 (Long-term Support): docs.unity3d.com/2017.4
Version 2017.3: docs.unity3d.com/2017.3
Version 2017.2: docs.unity3d.com/2017.2
Version 2017.1: docs.unity3d.com/2017.1

Unity 5 documentation

Version 5.6: docs.unity3d.com/560
Version 5.5: docs.unity3d.com/550
Version 5.4: docs.unity3d.com/540
Version 5.3: docs.unity3d.com/530
Version 5.2: docs.unity3d.com/520
Version 5.1: docs.unity3d.com/510
Version 5.0: docs.unity3d.com/500

Unity 4 documentation

Version 4.6: docs.unity3d.com/460
Version 4.5: docs.unity3d.com/450
Version 4.3: docs.unity3d.com/430
Version 4.2: docs.unity3d.com/420
Version 4.1: docs.unity3d.com/410
Version 4.0: docs.unity3d.com/400

Unity 3 documentation

Version 3.5.5: docs.unity3d.com/355
Version 3.5.3: docs.unity3d.com/353
Version 3.5.2: docs.unity3d.com/352
Version 3.5.1: docs.unity3d.com/351

Switching between Unity versions in the documentation

Leave feedback

The Unity Manual and Scripting API hold documentation for several versions of Unity. There are several ways to switch between versions.

Switch between the home pages of the two latest versions

You can switch between the two latest versions by clicking on the link at the top-left of every page, above the table of contents. These links always take you to the home page.

Version **2017.2** (switch to 2017.3b)

Example of the home page link at the top-left of every page

Switch versions and maintain the page

Version 5.6 and later

In Unity documentation versions from 5.6 onwards, you can look at an individual Unity documentation page and switch between the page versions using the **Other Versions** link on the right of every page.

Other Versions

The **Other Versions** link which maintains the page you are on

Sometimes, you might get a 404 error when you switch over. This happens if the page you're looking for doesn't exist in that version of the Manual or Scripting API. This is often the case if the feature isn't present in the corresponding version of Unity.

Version 5.5 and earlier

Documentation versions earlier than 5.6 do not have the **Other Versions** link.

To learn how to use a workaround for pre–5.6 documentation, follow these steps:

From the Manual or Scripting API home page, switch the version you are viewing using the home page link in the top-left corner of the page. Note that the structure of the URL changes. Each version has a specific URL:

Unity Manual

Destination	URL
Standard URL	https://docs.unity3d.com/Manual/index.html
Version-specific URL (2017.2)	https://docs.unity3d.com/2017.2/Documentation/Manual/
Version-specific URL (2017.1)	https://docs.unity3d.com/2017.1/Documentation/Manual/
Version-specific URL (5.6)	https://docs.unity3d.com/560/Documentation/Manual/

Unity Scripting API

Destination	URL
Standard URL	https://docs.unity3d.com/ScriptReference/index.html

Destination	URL
Version-specific URL (2017.2)	https://docs.unity3d.com/2017.2/Documentation/ScriptReference/
Version-specific URL (2017.1)	https://docs.unity3d.com/2017.1/Documentation/ScriptReference/
Version-specific URL (5.6)	https://docs.unity3d.com/560/Documentation/ScriptReference/

Now, navigate the Manual and Scripting API as usual. If you want to switch between versions, change the number in the URL to reflect which version you wish to view. This workaround works for all currently published versions of the documentation.

Sometimes, you might get a 404 error when you switch over. This happens if the page you're looking for doesn't exist in that version of the Manual or Scripting API. This is often the case if the feature isn't present in the corresponding version of Unity.

2017–07–07 Page amended with editorial review

Documentation Other Versions switcher added in 5.6

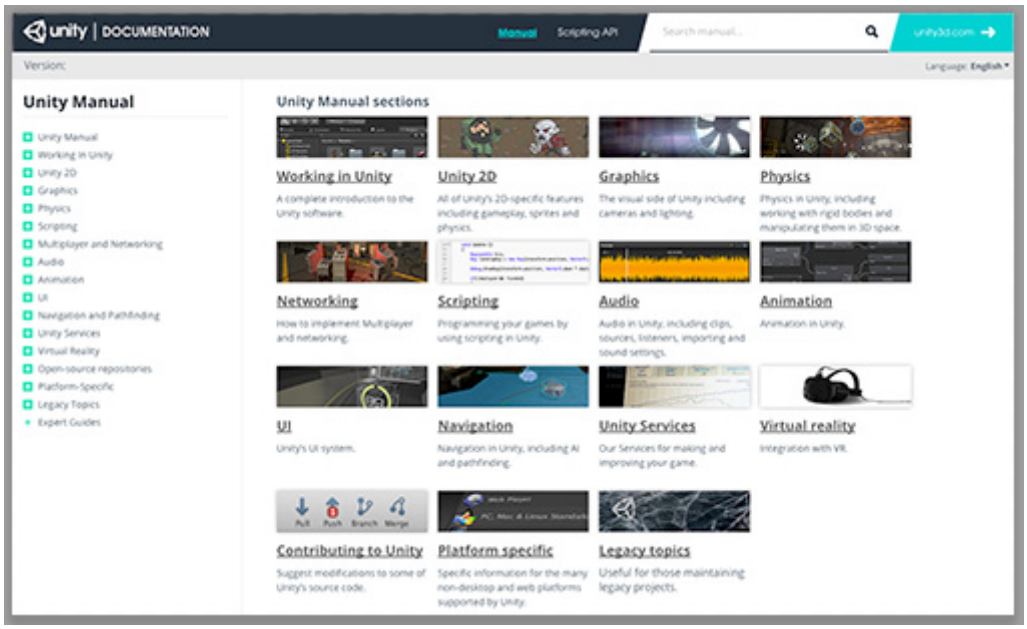
Offline documentation

Leave feedback

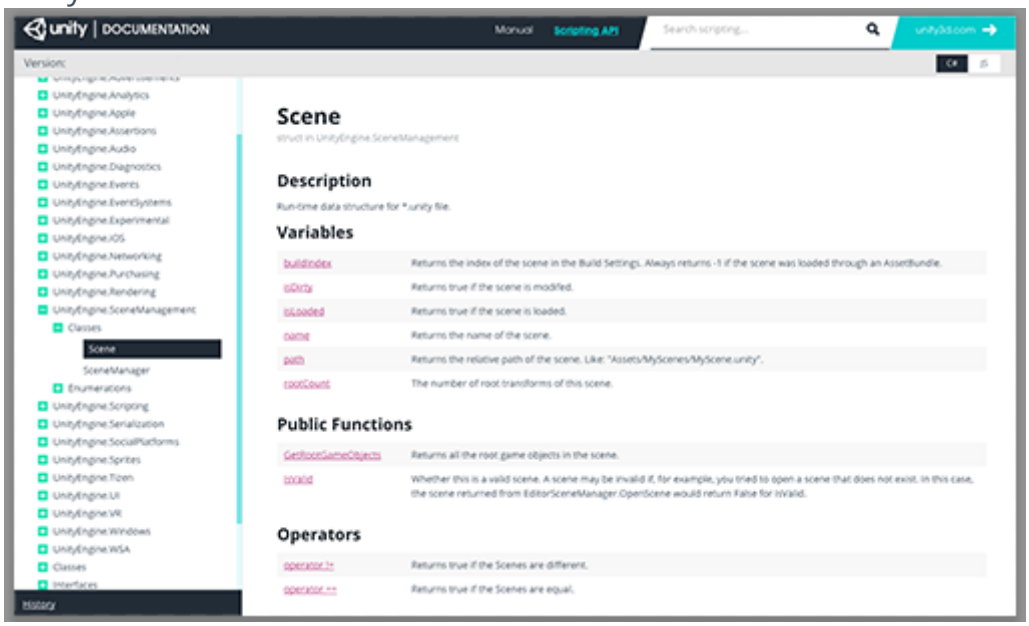
You can download the Unity Documentation as a zip file for offline use. To download the current version of the Unity Documentation, click the link below.

Download: Offline Unity Documentation (Size: about 300MB)

The zip file contains the most up-to-date version of the **Unity Manual** and **Unity Scripting API**.



Unity Manual



Unity Scripting API (Scripting Reference)

Documentation editorial review

Leave feedback

Some Unity User Manual pages contain a status referring to “editorial review”.

What does “editorial review” mean?

This is a documentation process where technical writers take raw notes from a developer, and work with the developer to make sure that documentation is correct, complete, well-written, and easy to follow.

“With no editorial review” means the page consists of ‘raw’ notes that we haven’t put through this process.
“With editorial review” means we have put the page through this process.
“With limited editorial review” means we have put some sentences or paragraphs on the page through this process, not the whole page.
For more information on technical documentation processes and techniques, see Wikipedia pages on Technical writing, Technical communication, and Technical editing.

Why does Unity publish documentation with no editorial review?

Editorial review takes time. We publish documentation with no editorial review in order to get information to Unity users as quickly as possible.

Are pages with no editorial review wrong?

Generally, no. They are the developer’s explanation, and so the most correct source of information we have. Pages without editorial review might be harder to read or understand, because they haven’t been edited to a technical documentation standard.

Why do some pages not have an editorial review status?

Only pages new or updated since the 8th May 2017 contain this information.

Do you want feedback on this?

Yes, please. You can rate our pages and provide other kinds of feedback using the **Feedback Form** at the bottom of every page.

Note that we can’t provide support via the documentation feedback form. Please send support questions and make bug reports through the usual channels.

2017–06–01 Page published with editorial review

Working in Unity

Leave feedback

This section provides a complete introduction to Unity:

Getting Started Downloading and installing Unity, getting set up to start your first project, and a quick tour of the Editor.

Asset Workflow How to get **assets** into Unity from a variety of different sources, including graphics, art and sound from external programs, Package files from other developers, and ready-made **Assets** from our **Asset Store** and the **Standard Assets** bundled with Unity.

The Main Windows A more in-depth look at each of the main windows you'll use every day in Unity, including useful shortcuts and hotkeys.

Creating Gameplay How to get started making **Scenes**, **GameObjects** and **Components**; reading input; and adding gameplay or interactivity to your Project.

Editor Features Information about many of the Editor's powerful features, to help you customize your workflow, integrate with external tools, and extend the Editor itself.

Advanced Development Information for experienced developers who want to take projects further using Plug-ins, AssetBundles, and other more advanced development techniques.

Advanced Editor Topics Take full control of the Editor, find out how it works under the hood, and learn how to script and customise the Asset pipeline and the Editor itself.

Licenses and Activation Understanding how to activate Unity and manage your licenses.

Upgrade Guides Important notes for upgrading projects that were authored with older versions of Unity.

Getting Started

Leave feedback

This section is your key to getting started with Unity. It explains the Unity interface, menu items, using **assets**, creating **scenes**, and publishing builds.

When you are finished reading this section, you will understand how Unity works, how to use it effectively, and the steps to put a basic game together.

Installation Options

You can install the Unity Editor in the following ways:

The Hub provides a central location to manage your Editor installations, Accounts and Licenses, and Projects. For more information on installing the Editor using the Hub, see Installing Unity using the Hub.
You can use the Download Assistant to install the Editor. If you subsequently choose to install the Hub, you can add it to the Hub at that time. For more information, see Installing Unity using the download assistant.
The Unity Download Assistant supports offline deployment. This allows you to download all the necessary files for installing Unity and generate a script to install the Editor on computers without internet access. For more information, see Installing Unity offline using the Download Assistant.

2018–06–12 Page amended with limited editorial review

Unity Hub

Leave feedback

The Unity Hub is a standalone application that streamlines the way you find, download, and manage your Unity Projects and installations. In addition, you can manually add versions of the Editor that you have already installed on your machine to your Hub.

You can use the Hub to:

Manage your Unity account and Editor licenses.

Create your Project, associate a default version of the Unity Editor with the Project, and manage the installation of multiple versions of the Editor.

Set your preferred version of Unity, but also easily launch other versions from your **Project view**.

Manage and select Project build targets without launching the Editor.

Run two versions of Unity at the same time. Note that to prevent local conflicts and other odd scenarios, you should only open a Project in one Editor instance at a time.

Add **components** to existing installations of the Editor. When you download a version of the Editor through Unity Hub, you can find and add additional components (such as specific platform support, Visual Studio, offline docs and Standard Assets) either during initial install or at a later date.

Use Project Templates to jump-start the creation process for common Project types. Unity's Project Templates provide default values for common settings when you create a new Project. Project Templates preset batches of settings for a target game type or level of visual fidelity. For more information, see Project Templates.

For information on installing the Unity Hub, see Installing Unity using the Hub.

2018–06–12 Page published with editorial review

Installing Unity using the Hub

Leave feedback

The Unity Hub is a management tool that allows you to manage all of your Unity projects and installations. Use the Hub to manage multiple installations of the Unity Editor along with their associated **components**, create new Projects, and open existing Projects.

Installing the Unity Hub

To install the Unity Hub, visit Download Unity Personal on the Unity website,

To install and use the Unity Editor, you must have a Unity Developer Network (UDN) account. If you already have an account, sign in and proceed to the Installing the Unity Editor section.

If you do not have an account, follow the prompts to create one. You can choose to create a Unity ID, or use one of the social sign-ins. For more information on accounts and subscriptions, see Unity Organizations.

Installing the Unity Editor

To install the Editor:

Click the **Installs** tab. The default install locations are:

Windows:

```
C:\Program Files\Unity\Hub\Editor
```

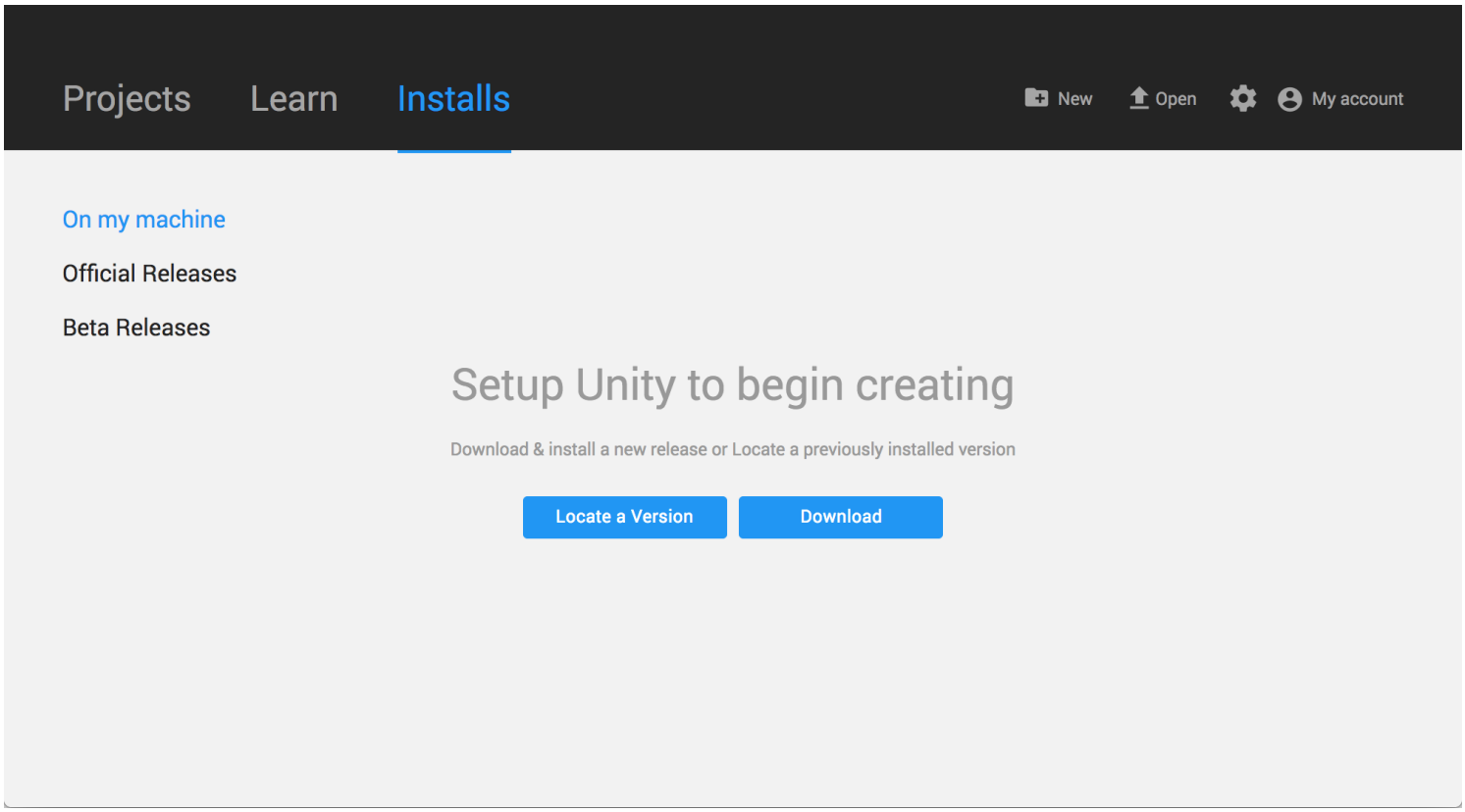
Mac:

```
/Application/Unity/Hub/Editor
```

Optionally, to change the default installation location, click the Gear icon.

In the **Editor Folder Location** dialog box, enter the new installation location and click **Done**.

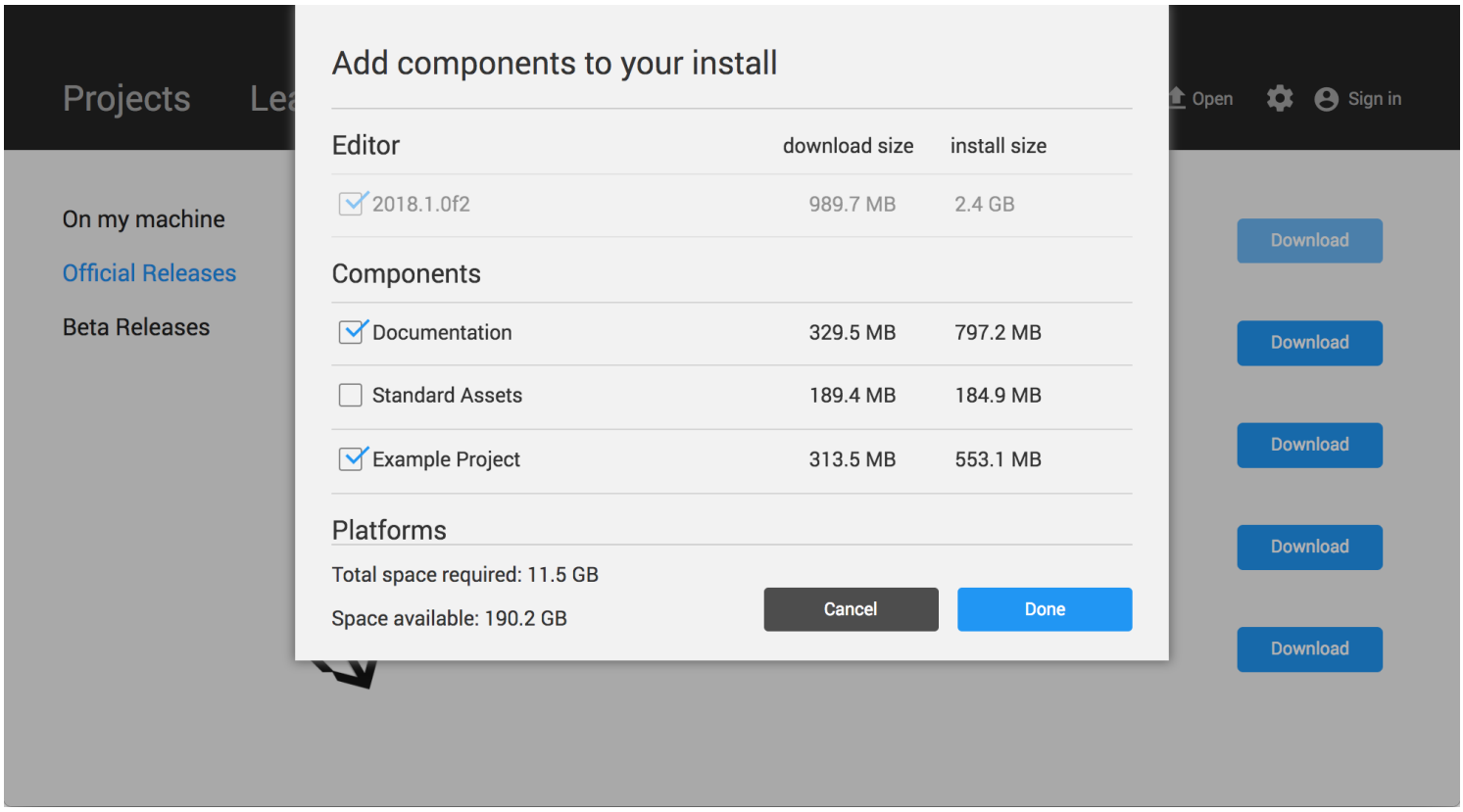
Click **Official Releases** for released versions of the Editor, or **Beta Releases** for the latest Beta release of the Editor.



Click the **Download** button of the Editor version you want to install. This opens a dialog box called **Add components to your install**.

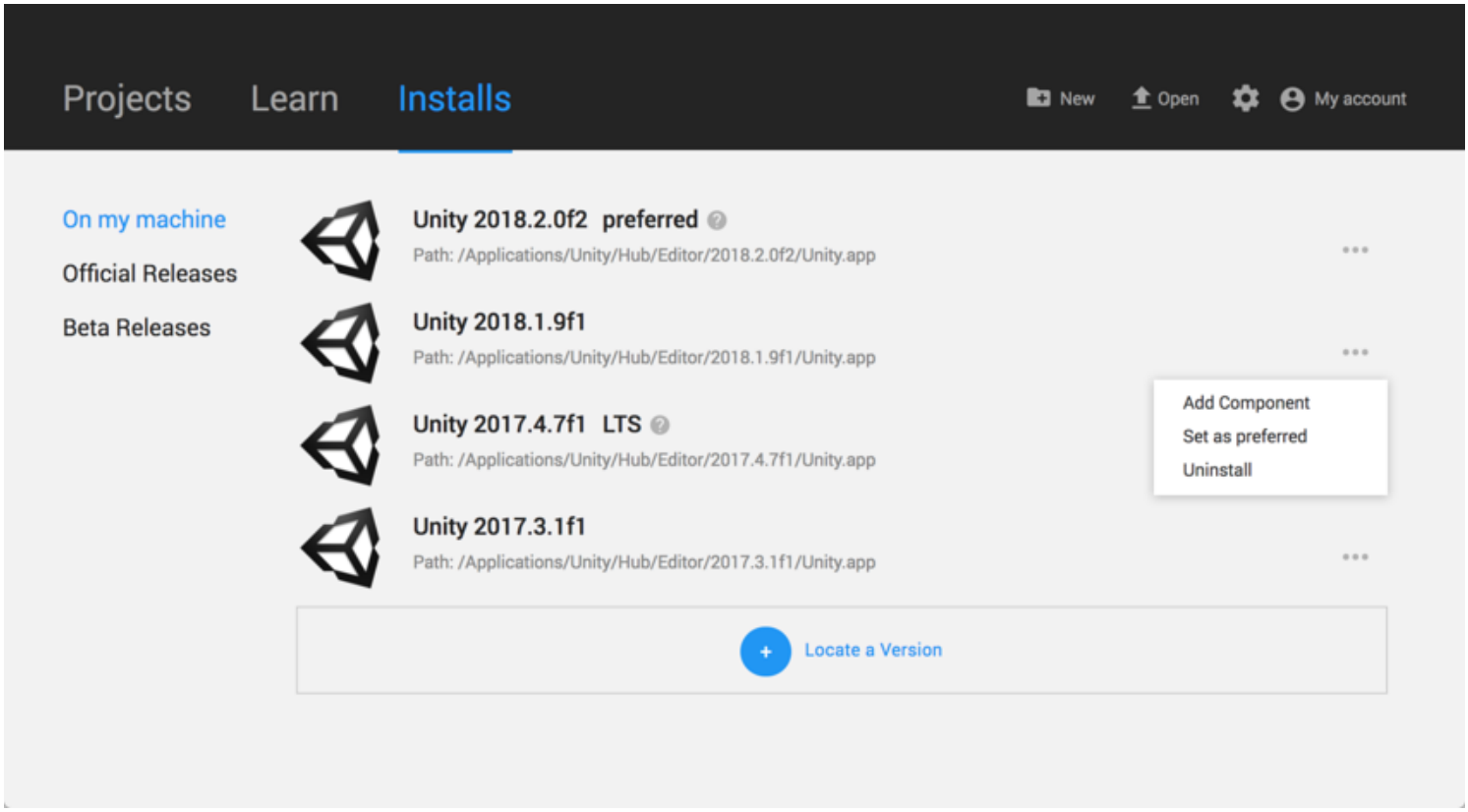
In the **Add components to your install** dialog box, select the components you want to install with the Editor, and click **Done**. If you don't install a component now, you can add it later if you need to.

If you are installing multiple Editor versions, the first installation starts as soon as the download is complete. Other selected versions download simultaneously and queue to start when the current installation finishes.



The Hub displays the installation location of each Editor under the corresponding version label.

To set it an Editor version as your preferred version, add components to it, or uninstall it, click the three dots next to that Editor version.



If you remove or uninstall the preferred Editor version, another installed Editor version becomes the preferred version.

Adding existing instances of the Editor to the Hub

You can add instances of the Editor to the Hub that you installed outside of the Hub.

Click the **Installs** tab.
Click the **On my machine** tab. To find existing installations of the Editor, click **Locate a Version**.
In the file dialog, navigate to the location of the Editor installation and select the Unity executable. On MacOS this is *Unity.app*. On Windows this is *Unity.exe*.
On macOS, the path is typically:

```
/Applications/Unity/Hub/Editor/<version>/Unity.app
```

On Windows, the path is typically:

```
C:\Program Files\Unity\Editor\Unity.exe
```

Or

```
C:\Program Files\Unity<version>\Editor\Unity.exe
```

Click the **Select editor** button.
To set the Editor as the preferred version, or to remove the Editor from the Hub, click the three dots next to the Editor version.

Removing an Editor that you added in this manner does not uninstall it or modify it in anyway.

Support for Editor versions prior to 2017.1

Sign in status is not shared for pre–2017.1 versions of the Editor opened through the Hub. Performing tasks such as **Manage License**, **Open Project**, **Create Project**, and **Sign in** opens the Unity Launcher instead of the Hub.

If you attempt to use the Unity Hub to open an Editor version 5 or earlier and you do not have an appropriate license file, the Editor will hang on the splash screen.

To avoid this issue, run the Editor directly, external to the Unity Hub, and the Editor will load correctly even if the license file is not detected.

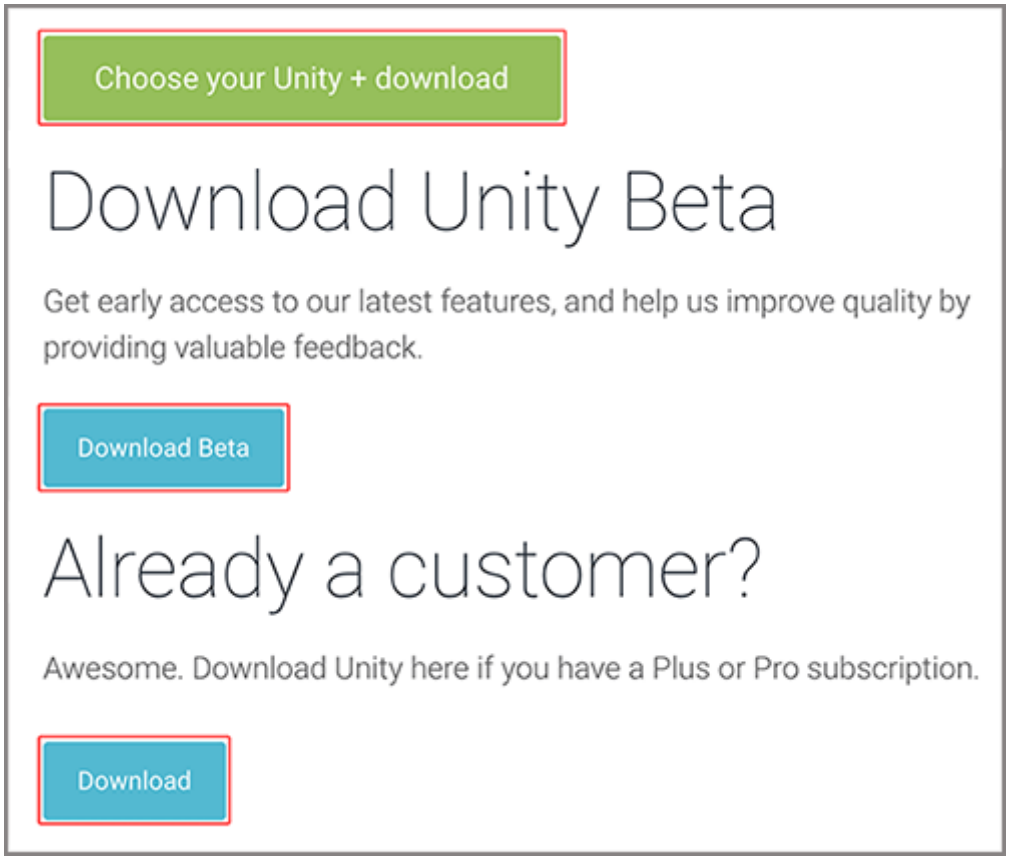
2018–06–12 Page published with editorial review

Installing Unity without the hub

Leave feedback

Download and install the Unity Editor from the Unity download page. This page gives you Unity Installer download links for both the latest full release version of Unity as well as the current Beta. If you require a Unity Plus or Pro license, you first need to confirm details for the license (number of seats, payment plan etc.).

The Unity download page presents you with the following options:



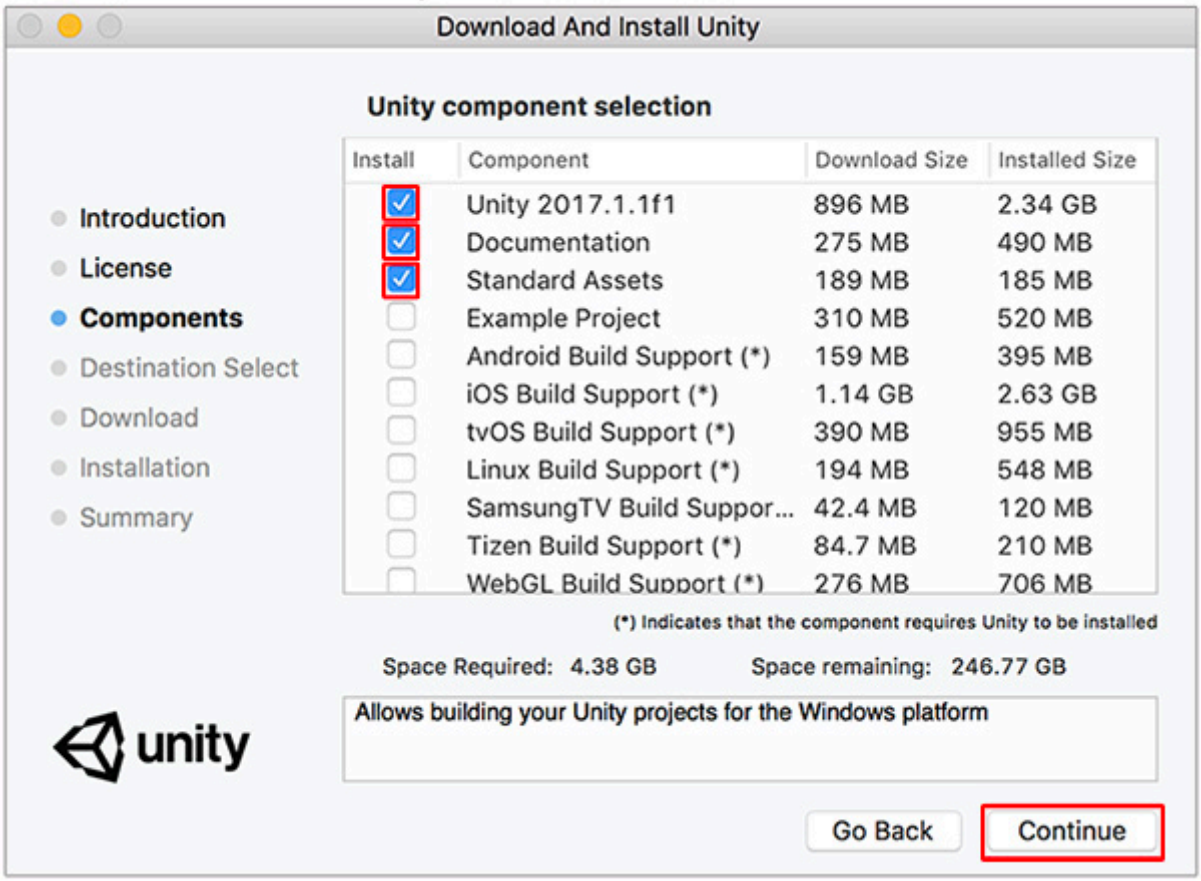
Unity Download page options

On the Unity download page, choose your desired version of the Unity Installer.

Unity installer

The Unity installer is a small executable program (approximately 1 MB in size) that lets you select which **components** of the Unity Editor you want to download and install.

If you're not sure which components you want to install, leave the default selections, click **Continue**, and follow the installer's instructions.



Unity installer (leave the default selected components if you're not sure which to choose)

Note: On PC there is also an extra option for Microsoft Visual Studio Community 2017.

Installing Unity without the Unity installer

If you prefer, you can download and install all of the components separately, without using the Unity installer. The components are normal installer executable programs and packages, so you may find it simpler to use the installer, especially if you are a new Unity user. Some users, such as those wishing to automate deployment of Unity in an organization, may prefer to install from the command line.

Installing Unity on Windows from the command line

If you want to automate deployment of Unity in an organization, you can install the Editor from the command line.

Use the following options when installing the Editor and other components from the command line on Windows.

Note: Installer command line arguments are case-sensitive.

Unity Editor install

Command	Details
/S	Performs a silent (no questions asked) install.
/D=PATH	Sets the default install directory. Useful when combined with the silent install option. Default folder is **C:\Program Files (x86)\Unity** (32-bit) or **C:\Program Files\Unity** (64-bit).

Example:

```
UnitySetup64.exe /S /D=E:\Development\Unity
```

This example installs Unity silently to a **E:\Development\Unity** folder, which will be the root of the Unity installation. In this case, the Editor executable will be installed in **E:\Development\Unity\Editor\Unity.exe**. The `/D` argument must be last and without quotes, even if the path contains spaces.

Unity Editor uninstall

To perform a silent uninstall, run `Uninstall.exe /S` from the ommand line or a script.

Note: Although the process finishes right away, it takes a few seconds before the files are actually removed. This is because the uninstaller is copied to a temporary location in order to be able to remove itself. Also, make sure the working directory is not inside the Unity install location, as it won't be able to remove the folder if this is the case.

Standard Assets install

To silently install **Standard Assets**:

```
UnityStandardAssetsSetup.exe /S /D=E:\Development\Unity
```

Note: If specifying a folder, use the Unity root folder (that is, the folder containing the Editor folder, and not where **Unity.exe** is installed into).

Example Project install

To silently install the Example Project, use:

```
UnityExampleProjectSetup.exe /S /D=E:\Development\Unity
```

Note: The default folder is `C:\Users\Public\Documentation\Unity Projects\Standard Assets Example Project`.

Installing Unity on OS X from the command line

The individual Unity installers are provided as .pkg files, which can be installed using the `installer` command, as described below.

Unity Editor install

To install the Editor into a `/Applications/Unity` folder on the specified target volume, enter:

```
sudo installer [-dumplog] -package Unity.pkg -target /
```

Standard Assets install

To install the Standard **Assets** into a `/Applications/Unity/Standard Assets` folder on the specified volume, enter:

```
sudo installer [-dumplog] -package StandardAssets.pkg -target /
```

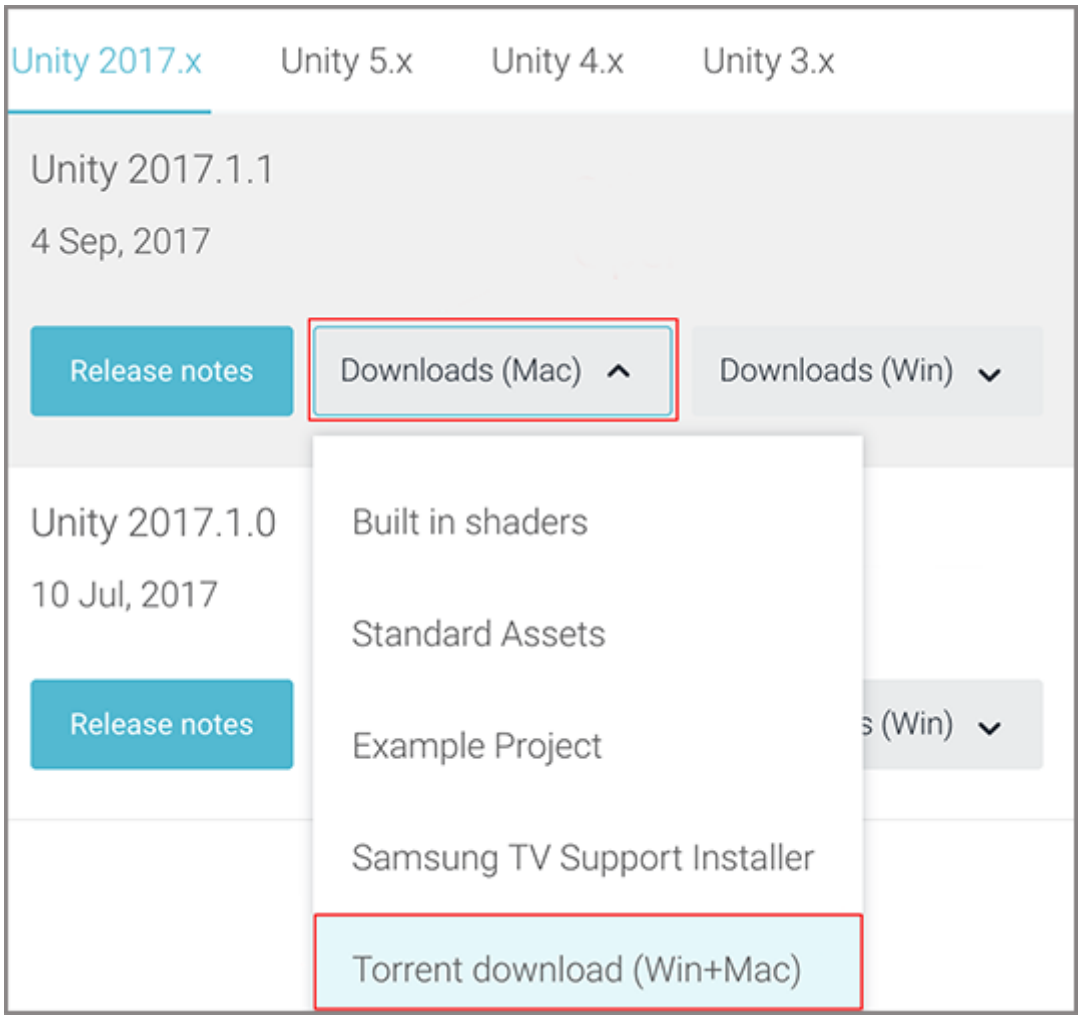
Example Project install

To install the Example Project into a `/Users/Shared/Unity/Standard-Assets` folder on the specified volume, enter:

```
sudo installer [-dumplog] -package Examples.pkg -target /
```

Torrent download

If you prefer to download Unity via a BitTorrent client, you can download get a torrent link from the Unity download archive page. Not all versions have a torrent download. If a version is available to download as a torrent, the option is presented as **Torrent download (Win+Mac)** in the **Downloads** dropdown menu.



Downloading Unity using a Torrent

Installing several versions at once

You can install as many versions of Unity as you like on the same computer.

On a Mac, the installer creates a folder called **Unity**, and overwrites any existing folder with this name. If you want more than one version of Unity on your Mac, rename the existing **Unity** folder before installing another version.

On a PC, the install folder is always named **Unity X.Y.Z[fp]W**, where the **f** is for an official release, and **p** is used to mark a patch release.

We strongly recommend that if you rename a Unity folder, you name the new folder logically (for example, add the version number to the end of the name). Note that any existing shortcuts, aliases and links to the offline docs may no longer point to the old version of Unity. This can be particularly confusing with the offline docs; if you suddenly find that browser bookmarks to the offline docs no longer work, then check that they have the right folder name in the URL.

2018–06–12 Page amended with editorial review

Installation advice updated in Unity 2017.2

Installation advice updated in Unity 2017.4

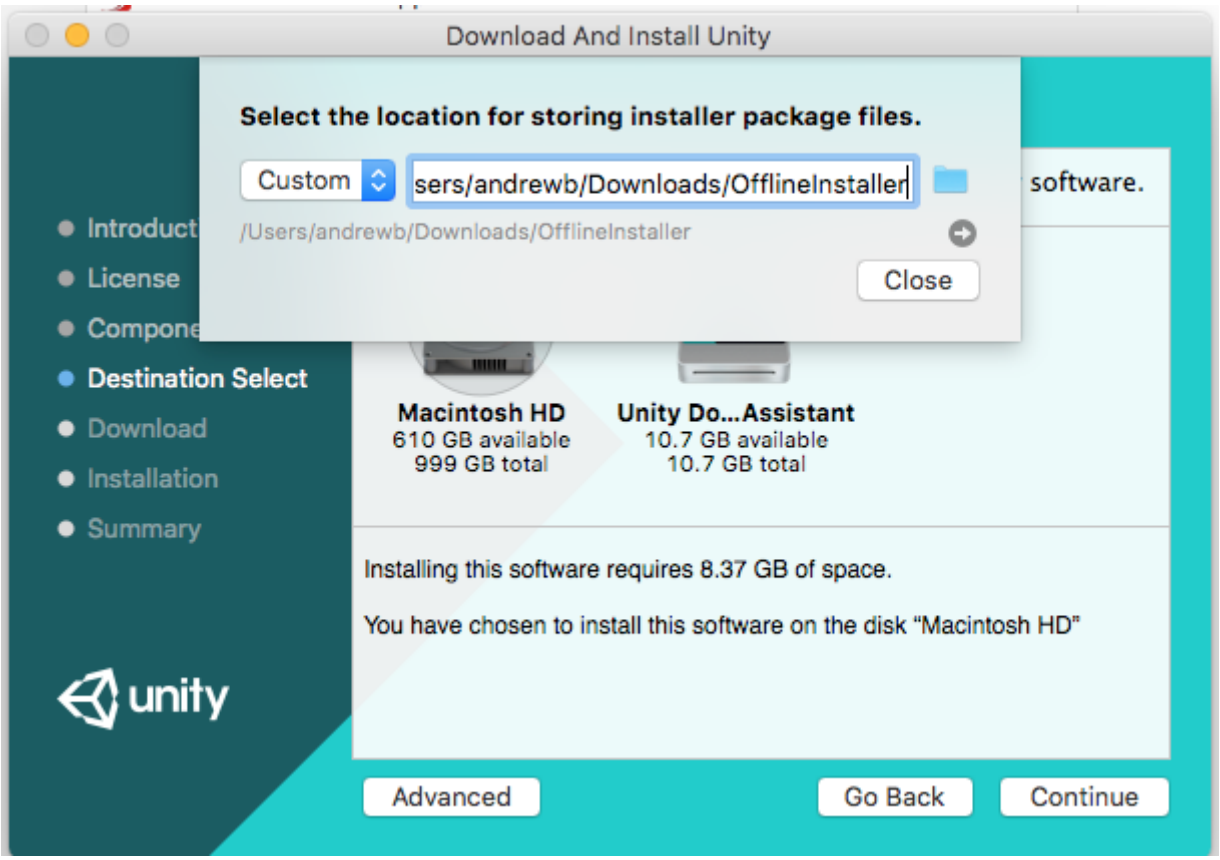
Installing Unity offline without the Hub

Leave feedback

The Unity Download Assistant supports offline deployment. This allows you to download all the necessary files for installing Unity, and to generate a script for repeating the same installation on other computers without internet access.

Preparation

Run the Download Assistant, and install Unity as normal on one computer. This computer must have enough free disk space to download all the files. Click the dropdown and select **Custom**, then choose the location you wish to download the files to.



Check you have everything you need

Open your PC's file manager, navigate to the custom location folder you specified earlier, and look for the `.sh` or `.bat` file inside that folder. Check the contents of this file. It should look similar to the following example:

install.sh

```
#!/bin/sh

# Run this script to automatically install or reinstall all packages,
# as it's performed by the Unity Download Assistant.
# Needs to be run with superuser permissions.

/usr/sbin/installer -pkg "Unity.pkg" -target "/" -verbose || exit 1
/usr/sbin/installer -pkg "Documentation.pkg" -target "/" -verbose || exit 1
/usr/sbin/installer -pkg "StandardAssets.pkg" -target "/" -verbose || exit 1
/usr/sbin/installer -pkg "Examples.pkg" -target "/" -verbose || exit 1
/usr/sbin/installer -pkg "UnitySetup-Android-Support-for-Editor-5.4.0b8.pkg" -target "/" -verbose || exit 1
/usr/sbin/installer -pkg "UnitySetup-iOS-Support-for-Editor-5.4.0b8.pkg" -target "/" -verbose || exit 1
/usr/sbin/installer -pkg "UnitySetup-WebGL-Support-for-Editor-5.4.0b8.pkg" -target "/" -verbose || exit 1
```

Deploying Unity to other computers

Windows

Copy the entire folder to the target Windows PC, and run the supplied `.bat` file.
To avoid the Windows UAC prompt, run `install.bat` from the Administrator shell. In the **Start** menu, search for `cmd.exe`, right-click, and select **Run as administrator**.
Navigate to the folder with the **scripts**. This will usually be in your Downloads folder (`cd C:\Users\[YourName]\Download\UnityPackages`).

Mac

Copy the entire folder to the target Mac OS X machine, and run the supplied `.sh` file. Run `sudo install.sh`.
Navigate to the folder with the scripts. This will usually be in your Downloads folder (`cd ~/Downloads/UnityPackages`).
You can repeat these instructions as many times as you need to for each computer you wish to install Unity on.

Unity Hub advanced deployment considerations

Leave feedback

This topic provides information to help you plan and configure your Unity Hub deployment to work with your choice of deployment system (for example, Microsoft SCCM). It covers:

Firewall configuration
Silent install and uninstall on Windows
Determining whether Unity Hub is installed on Windows

Firewall configuration

The Unity Hub uses the following ports:

Outbound TCP Port 443 (HTTPS): The Hub uses this port to connect to Unity Services, providing functionality such as user login, Unity Editor updates, Unity Collaborate, and updates for the Hub itself. If you do not open this port, the Unity Hub is still usable, but online features are disabled.
Outbound TCP/UDP Port 53 (DNS): The Hub uses this port to quickly determine whether an internet connection is available. If you do not open this port, the Hub assumes that no internet connection is available, even if Port 443 is open.
Inbound TCP port 39000, accepting connections from localhost only: The Hub listens on this port to accept connections from Editor instances running on the local machine. This port must be open for connections from localhost.

Silent install and uninstall on Windows

You can install and uninstall the Hub from the command line using the silent-mode parameter. Silent-mode installs the Hub without prompting for user input. Local administrator access is required, because the installation modifies the Program Files folder.

To install the Unity Hub using silent-mode:

Download the Unity Hub.
Open a command prompt and navigate to the folder in which you downloaded the Hub.
At the command line, run the following command, in which the `/S` initiates a silent installation:

```
UnityHubSetup.exe /S
```

To uninstall the Unity Hub silently, pass the `/S` command-line switch to the uninstaller:

Open a command prompt.
At the command line, run the following command, in which the /S initiates a silent uninstall:

```
"C:\Program Files\Unity Hub\Uninstall Unity Hub.exe" /S
```

Determining whether Unity Hub is installed on Windows

To determine if the Unity Hub is installed, check for the presence of the following registry key:

```
HKEY_LOCAL_MACHINE\Software\Unity Technologies\Hub
```

If this key is present in the registry, the Unity Hub is installed.

2018–06–15 Page published with editorial review

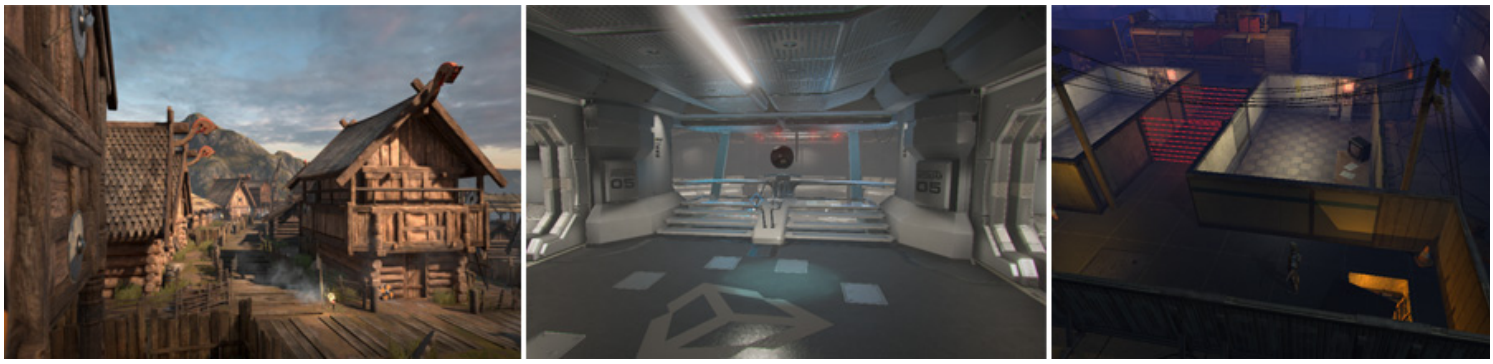
2D or 3D projects

Leave feedback

Unity is equally suited to creating both 2D and 3D games. When you create a new project in Unity, you have the choice to start in 2D or 3D mode. You may already know what you want to build, but there are a few subtle points that may affect which mode you choose.

The choice between starting in 2D or 3D mode determines some settings for the Unity Editor, such as whether images are imported as textures or **sprites**. You can swap between 2D or 3D mode at any time regardless of the mode you set when you created your project (see 2D and 3D Mode Settings). Here are some guidelines which should help you choose.

Full 3D



Some 3D scenes from Unity's sample projects on the Asset Store

3D games usually make use of three-dimensional geometry, with Materials and Textures rendered on the surface of **GameObjects** to make them appear as solid environments, characters and objects that make up your game world. The Camera can move in and around the **Scene** freely, with light and shadows cast around the world in a realistic way. 3D games usually render the **Scene** using perspective, so objects appear larger on screen as they get closer to the camera. For all games that fit this description, start in **3D** mode.

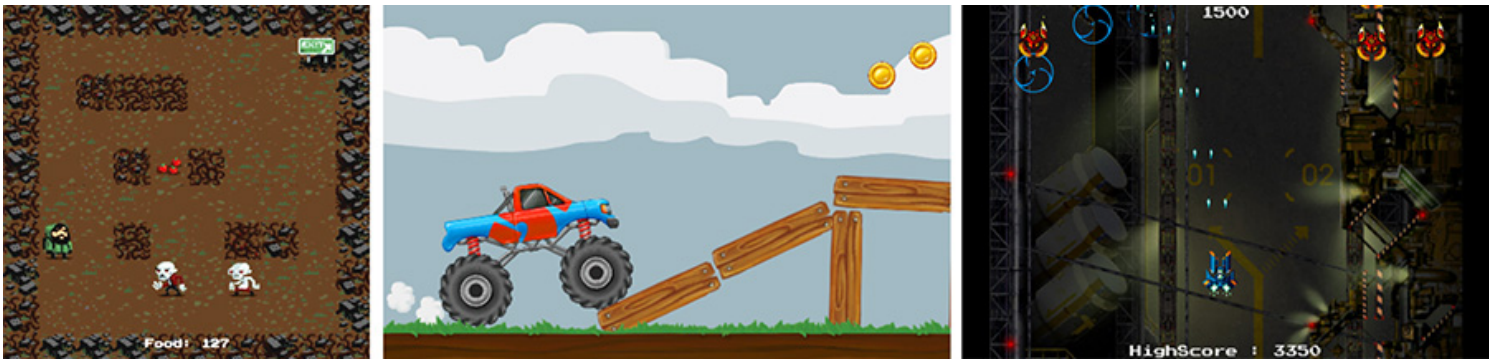
Orthographic 3D

Some 3D games using an Orthographic view

Sometimes games use 3D geometry, but use an orthographic camera instead of perspective. This is a common technique used in games which give you a bird's-eye view of the action, and is sometimes called "2.5D". If you're making a game like this, you should also use the Editor in **3D** mode, because even though there is no *perspective*, you will still be working with 3D models and **Assets**. You'll need to switch your Camera and Scene view to **Orthographic** though.

Scenes above from Synty Studios and BITGEM.

Full 2D



Some examples of typical 2D game types

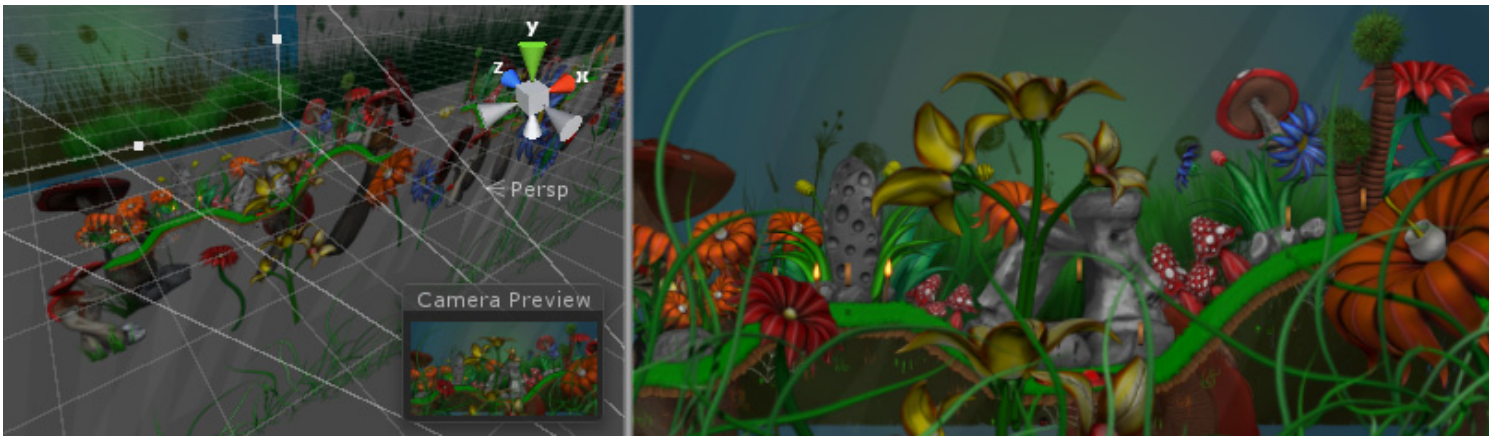
Many 2D games use flat graphics, sometimes called sprites, which have no three-dimensional geometry at all. They are drawn to the screen as flat images, and the game's camera has no perspective. For this type of game, you should start the editor in **2D** mode.

2D gameplay with 3D graphics

A side scrolling game with 2D gameplay, but 3d graphics

Some 2D games use 3D geometry for the environment and characters, but restrict the *gameplay* to two dimensions. For example, the camera may show a side-scrolling view, and the player can only move in two dimensions, but the game itself still uses 3D models for the obstacles and a 3D perspective for the camera. For these games, the 3D effect may serve a stylistic rather than functional purpose. This type of game is *also* sometimes referred to as "2.5D". Although the gameplay is 2D, you are mostly manipulating 3D models to build the game, so you should start the editor in **3D** mode.

2D gameplay and graphics, with a perspective camera



A 2D "cardboard theatre" style game, giving a parallax movement effect

This is another popular style of 2D game, using 2D graphics but with a perspective camera to get a parallax scrolling effect. This is a "cardboard theater"-style scene, where all graphics are flat, but arranged at different distances from the camera.

It's most likely that **2D** mode will suit your development in this case. However, you should change your Camera's projection mode to **Perspective** and the Scene view mode to **3D**.

Scene above from One Point Six Studio.

Other styles

You may have plans for a project that fits one of the above descriptions, or you may have something else entirely different or unique in mind. Whatever your plans are, hopefully the above will give you an idea of which mode to start the Editor in. Remember, you can switch modes at any time later.

See 2D and 3D Mode Settings to learn more about how to change the 2D/3D mode, and find more detail about how the modes differ.

Useful 2D project information

Whichever type of project you are working in, there are some useful pages to help you get started. There are also many specific pages for 2D features. See the Unity 2D section of the Unity User Manual.

Getting started with Unity

Unity Basics
Creating Scenes
Creating Gameplay

2D Development with Unity

Unity 2D Manual Documentation

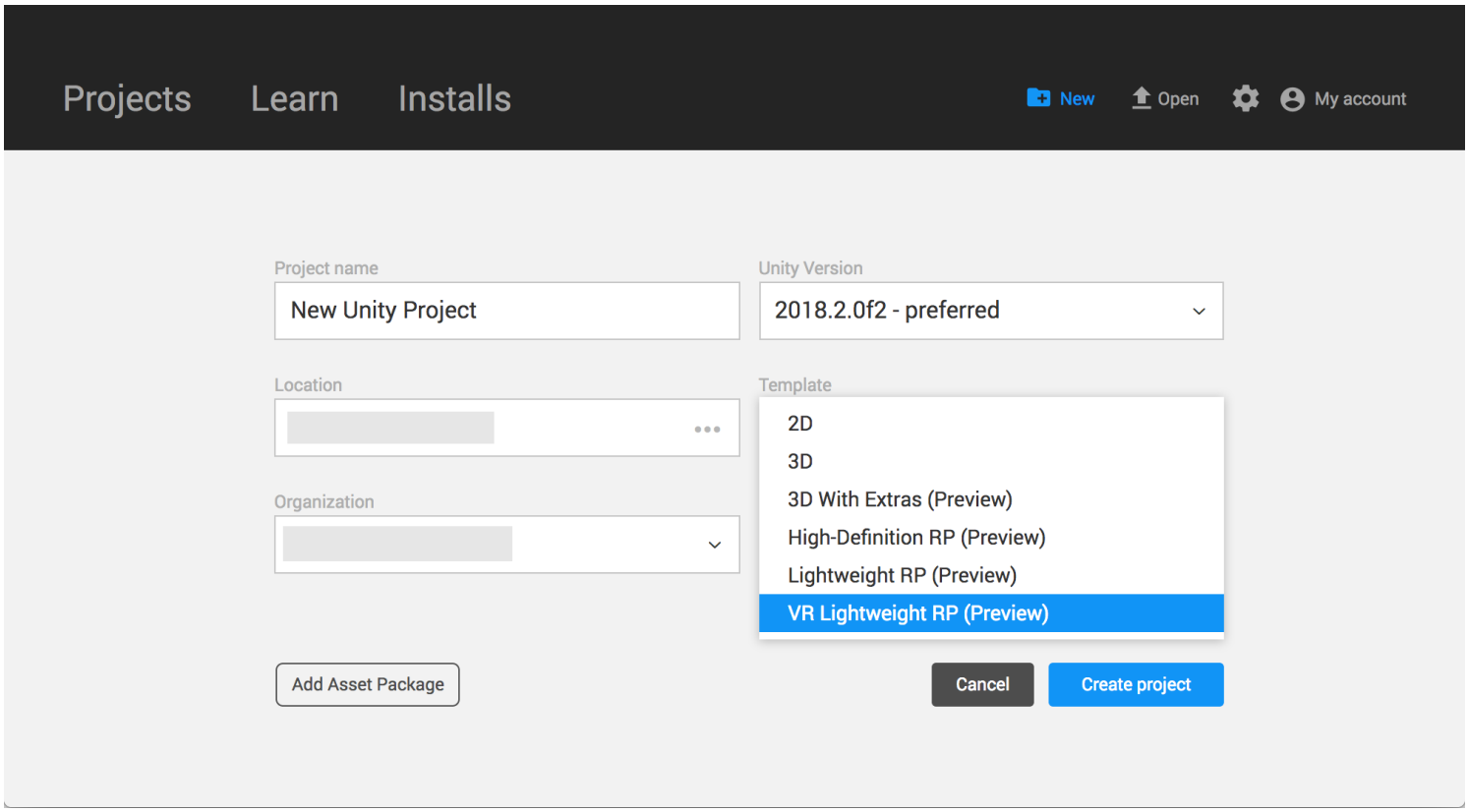
Project Templates

Leave feedback

Project Templates provide preselected settings based on common best practices for different types of Projects. These settings are optimized for 2D and 3D Projects across the full range of platforms that Unity supports.

Templates speed up the process of preparing the initial Project, a target game-type or level of visual fidelity. Using Templates introduces you to settings that you might not have discovered, and to features such as Scriptable Render Pipelines, Shader Graph, and Post Processing.

When you create a Project, you select a Template with which to initialize your Project.



Project Templates drop-down selection

Template types

Unity provides the following templates, which you can use to initialize your Project.

2D

Configures **Project settings** for 2D apps, including Texture (Image) Import, Sprite Packer, Scene View, Lighting, and Orthographic Camera.

3D

Configures Project settings for 3D apps that use Unity's built-in rendering pipeline.

3D With Extras (Preview)

Configures Project settings for 3D apps that use Unity's built-in renderer and post-processing features. This Project type includes the new post-processing stack, several Presets to jump-start development, and example content.

For more information on post-processing, see documentation on post-processing in the post-processing GitHub repository.

High Definition RP (Preview)

Configures Project settings for Projects that use high-end platforms that support **Shader** Model 5.0 (DX11 and above). This template is built using the Scriptable Render Pipeline (SRP), a modern **rendering** pipeline that includes advanced material types and a configurable hybrid tile/cluster deferred/forward lighting architecture. This Template also includes the new post-processing stack, several Presets to jump start development, and example content.

This Template adds the following features to your Project:

SRP - For more information, see documentation on the Scriptable Render Pipeline in the SRP GitHub repository.
Post-Processing stack - The post-processing Stack enables artists and designers to apply full -screen filters to **scenes** using an artist-friendly interface. For more information, see documentation on post-processing in the post-processing GitHub repository.
Note: The High Definition RP is currently in development, so consider it incomplete and subject to change (API, UX, scope). As such, it is not covered by regular Unity support. Unity is seeking feedback on the feature. To ask questions about the feature, visit the Unity preview forum.

Lightweight RP (Preview)

Configures Project settings for Projects where performance is a primary consideration and projects that use a primarily baked lighting solution. This template is built using the Scriptable Render Pipeline (SRP), a modern rendering pipeline that includes advanced material types and a configurable hybrid tile/cluster deferred/forward lighting architecture. This Template also includes the new post-processing stack, several Presets to jump start development, and example content.

Using the Lightweight pipeline decreases the draw call count on your project, providing a solution for lower-end hardware.

This Project Template uses the following features:

SRP - For more information, see documentation on the Scriptable Render Pipeline in the SRP GitHub repository.
Shader Graph tool - This tool allows you to create shaders using a visual node editor instead of writing code. For more information, see documentation on the Shader Graph in the Shader Graph GitHub repository.
Post-processing stack - The post-processing Stack enables artists and designers to apply full -screen filters to scenes using an artist-friendly interface. For more information, see documentation on post-processing in the post-processing GitHub repository.
Note: The Lightweight RP is currently in development, so consider it incomplete and subject to change (API, UX, scope). As such, it is not covered by regular Unity support. Unity is seeking feedback on the feature. To ask questions about the feature, visit the Unity preview forum.

[XR] VR Lightweight RP (Preview)

Configures Project settings for Projects where performance is a primary consideration for **Virtual Reality** (VR) Projects that use a primarily baked lighting solution. This template is built using the Scriptable Render Pipeline (SRP), a modern rendering pipeline that includes advanced material types and a configurable hybrid tile/cluster deferred/forward lighting architecture. This Template also includes the new post-processing stack, several Presets to jump start development, and example content.

To use this Project, you need a device. Ensure that you have the correct SDKs for the device for which you are developing before you use this Template.

This Project Temple uses the following features:

VR - Unity VR lets you target virtual reality devices directly from Unity, without any external plug-ins in projects. For more information, see VR overview.
SRP - For more information, see documentation on the Scriptable Render Pipeline in the SRP GitHub repository.
Shader Graph tool - This tool allows you to create shaders using a visual node editor instead of writing code. For more information on the shader graph tool, see documentation on the Shader Graph in the Shader Graph GitHub repository.
Post-processing stack - The post-processing Stack enables artists and designers to apply full -screen filters to scenes using an artist-friendly interface. For more information, see documentation on post-processing in the post-processing GitHub repository.

Note: The Lightweight RP is currently in development, so consider it incomplete and subject to change (API, UX, scope). As such, it is not covered by regular Unity support. Unity is seeking feedback on the feature. To ask questions about the feature, visit the Unity preview forum.

2018–05–22 Page published with editorial review

Added Project Templates in 2018.1

New Project Templates added in 2018.2

Starting Unity for the first time

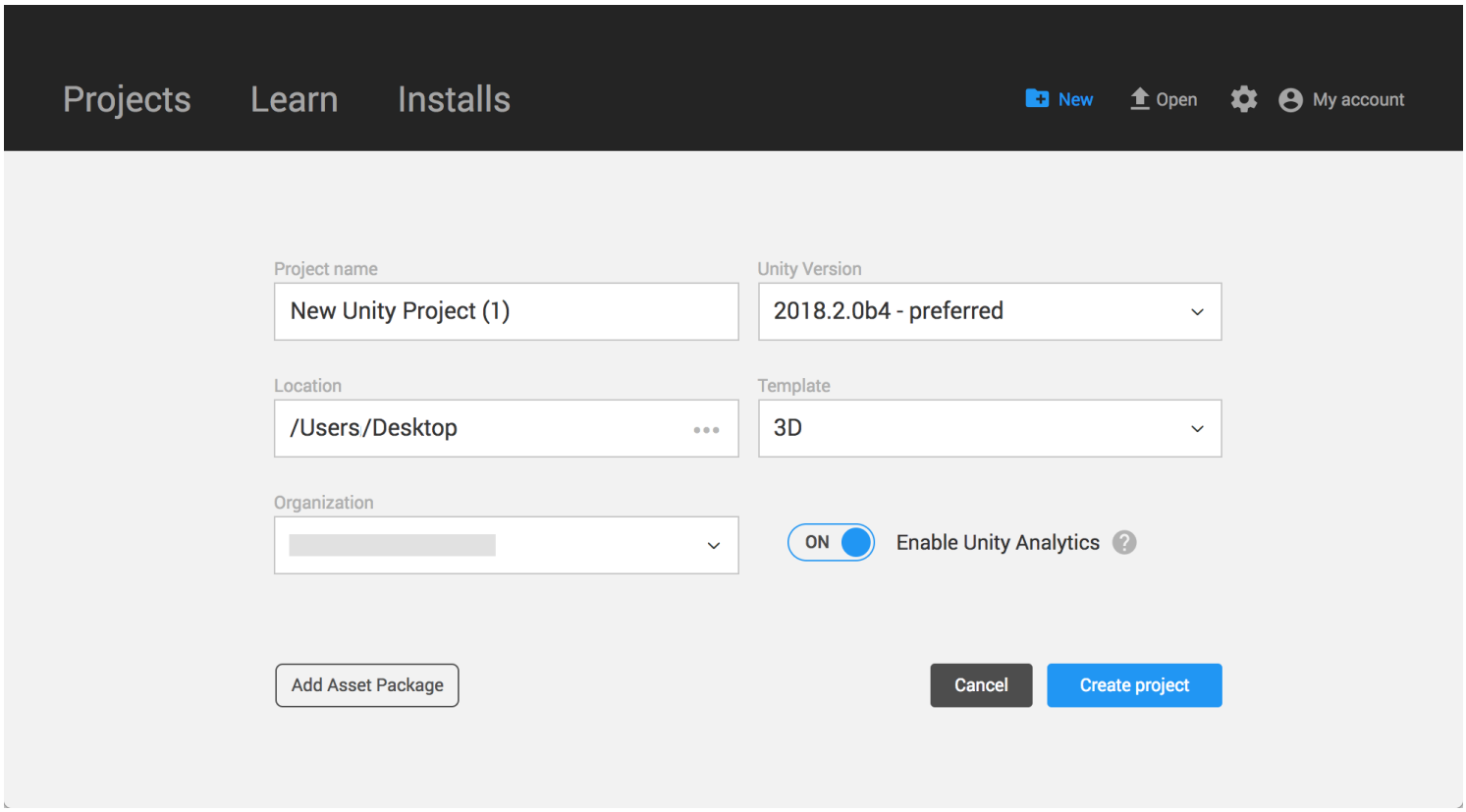
Leave feedback

There are two ways to create a new Project:

Using the Unity Hub - The Unity Hub is a standalone application that streamlines the way you find, download, and manage your Unity projects and installations. For information on installing the Unity Hub, see Installing Unity using the Hub.
Using the Unity Launcher - For more information on installing the Unity Editor with the Unity Launcher download assistant, see Installing Unity using the download assistant.

Creating a Project using the Hub

To create a new Project (and specify which Editor version to open it in), click **New**.



Create a Project in the Hub

Setting	Function
Project name	Sets the name of your Project. This names the main Project folder, which stores the Assets, **Scenes**, and other files related to your Project.
Unity Version	Select the Editor version you want to use to create the Project. **Note**: The drop-down menu is only available if you have installed multiple versions of the Editor in the Hub.
Location	Use this to define where in your computer's file system to store your Project. The location of your Project defaults to the home folder on your computer. To change it, type the file path to your preferred storage location into the Location field. Alternatively, click the ellipsis icon in the Location field. This opens your computer's file browser (Explorer, Finder or Files, depending on your computer's operating system). In your file browser, navigate to the folder that you want to store your Project in, and click Select Folder or Open.
Template	Choose a Project Template. Project Templates provide pre-selected settings based on common best practices for Projects. These settings are optimized for 2D and 3D Projects across the full range of platforms that Unity supports. The default Template type is 3D.

Setting	Function
Add Asset Package	Use this to add pre-made content to your Project. The Asset Packages provided with Unity include pre-made models, particle effects, and example scripts, along with other useful tools and content. To import Unity-provided Asset Packages into your Project, click the Add Asset Package button, then tick the checkbox to the left of each Asset Package you want to import, and click Done. Unity automatically imports the selected Assets when your Project is created. The Add Asset Package screen also contains any Assets you have downloaded from the Unity Asset Store. You can also add Asset Packages later, once you've created your project. To do this in the Unity Editor, go to **Assets > Import Package**, and select the package you want to import.
Enable Unity Analytics	Select whether to enable Unity **Analytics**. Unity **Analytics** is a data platform that provides **analytics** for your Unity game. Using **Analytics**, you can find out who the players are in your game and their in-game behavior. Unity **Analytics** are enabled by default.

Creating a Project using the Unity Launcher

To use the Unity Editor you must have a Unity Developer Network (UDN) account. If you already have an account, sign in and proceed to the The Projects tab section.

If you do not have an account, follow the prompts to create one. For more information on accounts and subscriptions, see Unity Organizations.

When you launch the Unity Editor, the Unity launcher Home Screen appears.

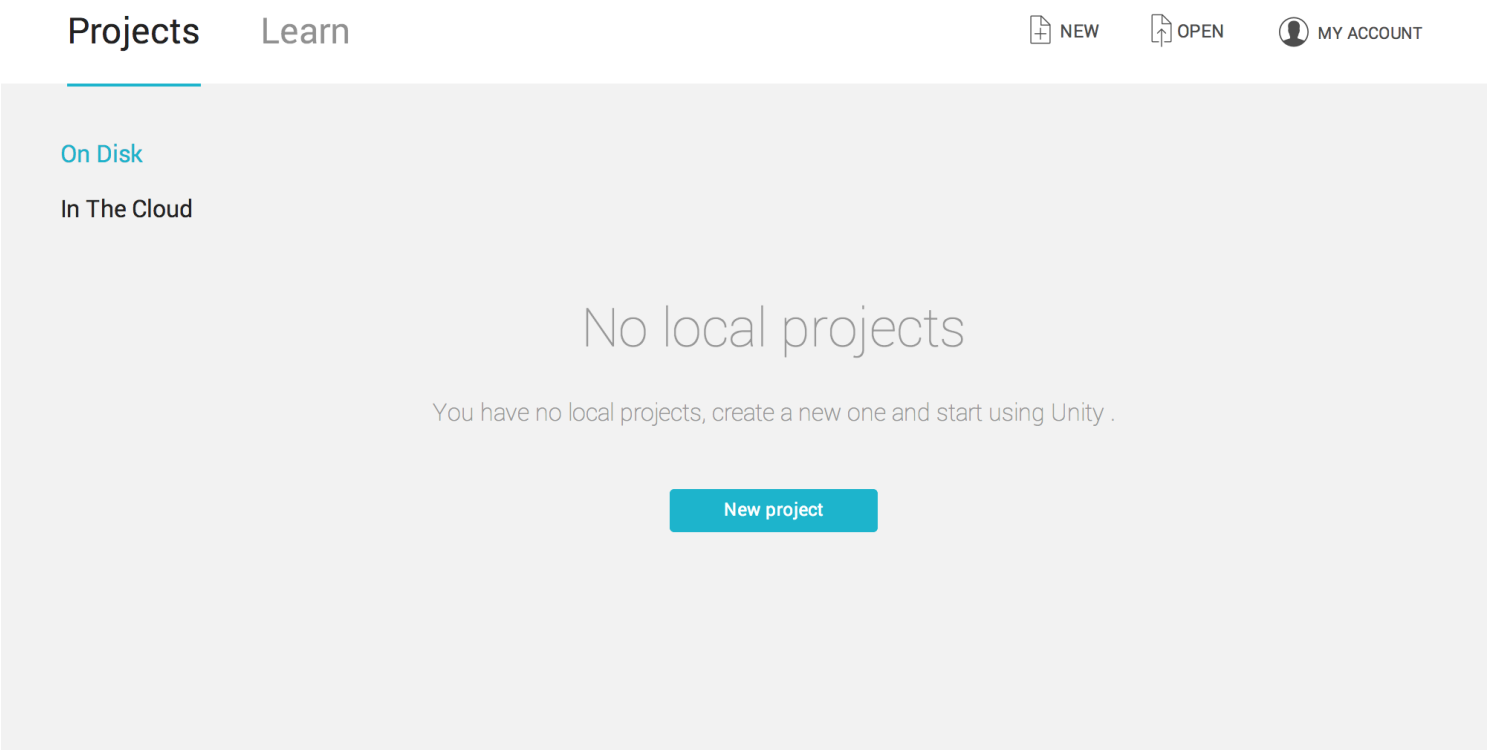
There are two tabs: The **Projects** tab and the **Learn** tab.

The **Projects** tab is where you can create new Projects or open existing Projects.

From the **Learn** tab, you can access tutorials and learning resources to help you get started with Unity. If you are new to using Unity, you can work through the Unity Learn tutorials before starting a new project.

The Projects tab

On the Home Screen, click **Projects** to view the contents of the **Projects** tab.



The Project tab in the Home Screen

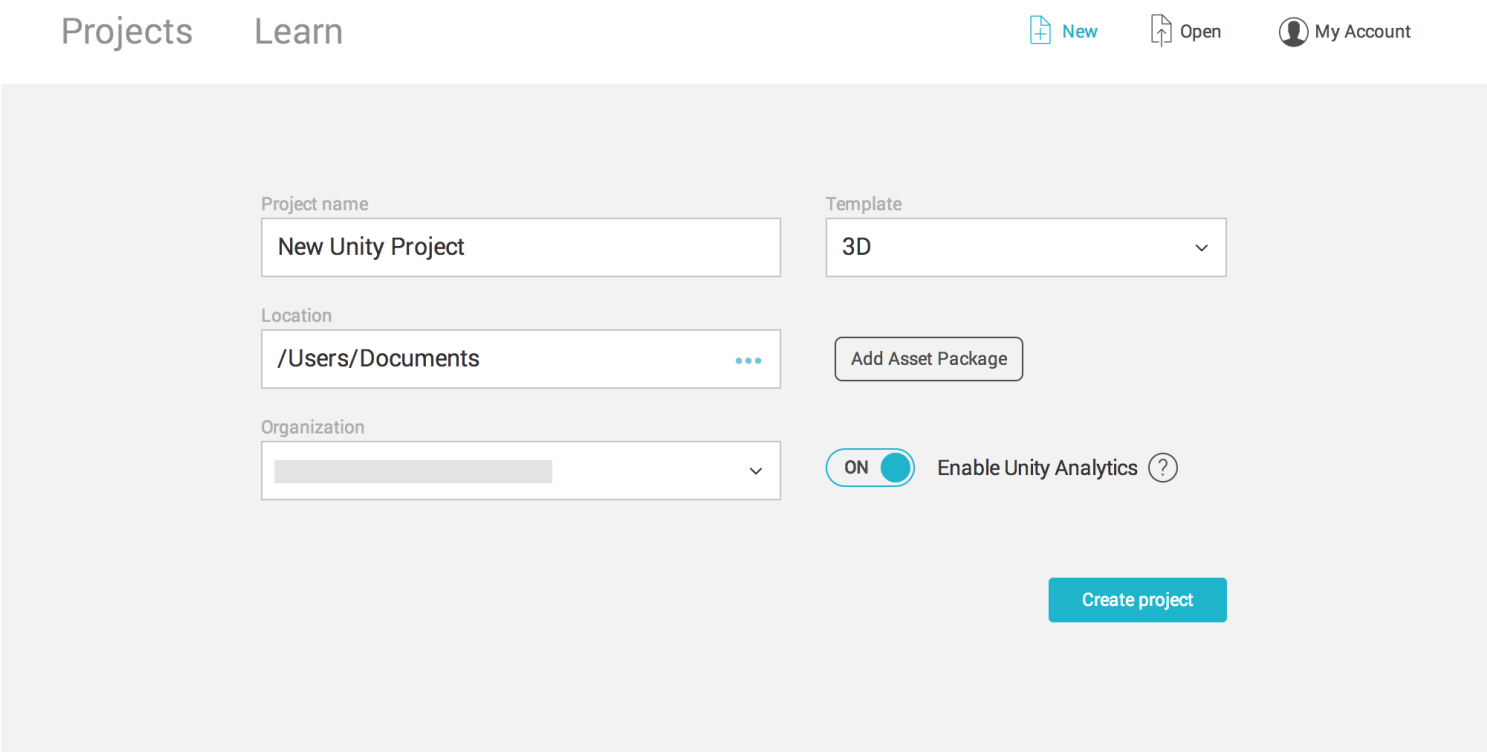
Unity stores Projects in two locations:

On Disk: To open an existing Unity Project stored on your computer, click the Project name in the **Projects** tab, or click **Open** to browse your computer for the Project folder.
In the Cloud: To access Unity Collaborate Projects, click **In the Cloud**, then select the Project you want to load. The Hub prompts you to choose a storage location for the Project on your computer.

Creating a Project

In the top right corner of the Home Window, click **New** to open the Create **Project View**.

From the Create Project View, there are **Project settings** to complete before Unity creates your project. These are described in detail below.



The Home Screen's Create Project window

Setting	Function
Project name	Sets the name of your Project. This names the main Project folder which stores the Assets, Scenes, and other files related to your Project.
Location	Use this to define where in your computer's file system to store your Project. The location of your Project defaults to the home folder on your computer. To change it, type the file path to your prefered storage location into the Location field. Alternatively, click the ellipsis icon in the Location field. This opens your computer's file browser (Explorer, Finder or Files, depending on your computer's operating system). In your file browser, navigate to the folder that you want to store your Project in, and click Select Folder or Open.
Template	Choose a Project Template. Project Templates provide pre-selected settings based on common best practices for Projects. These settings are optimized for 2D and 3D Projects across the full range of platforms that Unity supports. The default Template type is 3D.
Add Asset Package	Use this to add pre-made content to your Project. The Asset Packages provided with Unity include pre-made models, particle effects, and example scripts, along with other useful tools and content.To import Unity-provided Asset Packages into your Project, click the Add Asset Package button, then tick the checkbox to the left of each Asset Package you want to import, and click Done. Unity automatically imports the selected Assets when your Project is created. The Add Asset Package screen also contains any Assets you have downloaded from the Unity Asset Store.You can also add Asset Packages later, once you've created your project. To do this in the Unity Editor, go to **Assets** > **Import Package**, and select the package you want to import.
Enable Unity Analytics	Select whether to enable **Unity Analytics**. **Unity Analytics** is a data platform that provides analytics for your Unity game. Using Analytics, you can find out who the players are in your game and their in-game behavior. **Unity Analytics** are enabled by default.

When you are done, click the **Create Project** button. Unity automatically generates the required files, creates your Project and opens the Project.

2018–06–12 Page published with editorial review

The Learn tab

Leave feedback

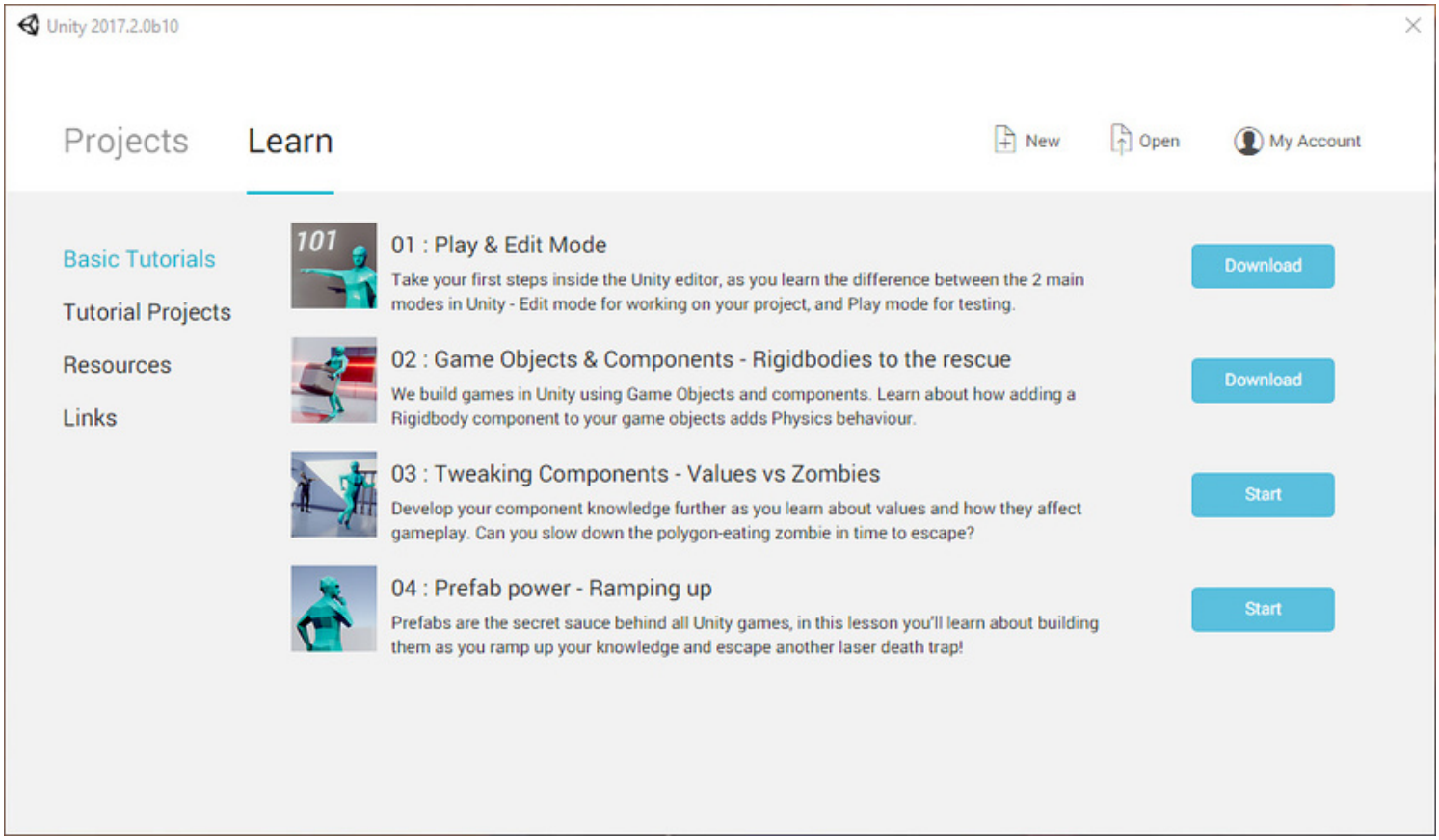
The **Learn** tab in the Home window gives you access to a variety of tutorials and learning resources (including example Projects that you can import directly into Unity) to help you get started with Unity.

The Learn tab is visible in the Home window after Unity is opened. The Home window can also be accessed from within the Editor by navigating to **File** > **New Project**.

The Learn tab is split into four sections:

Basic Tutorials
Tutorial Projects
Resources
Links

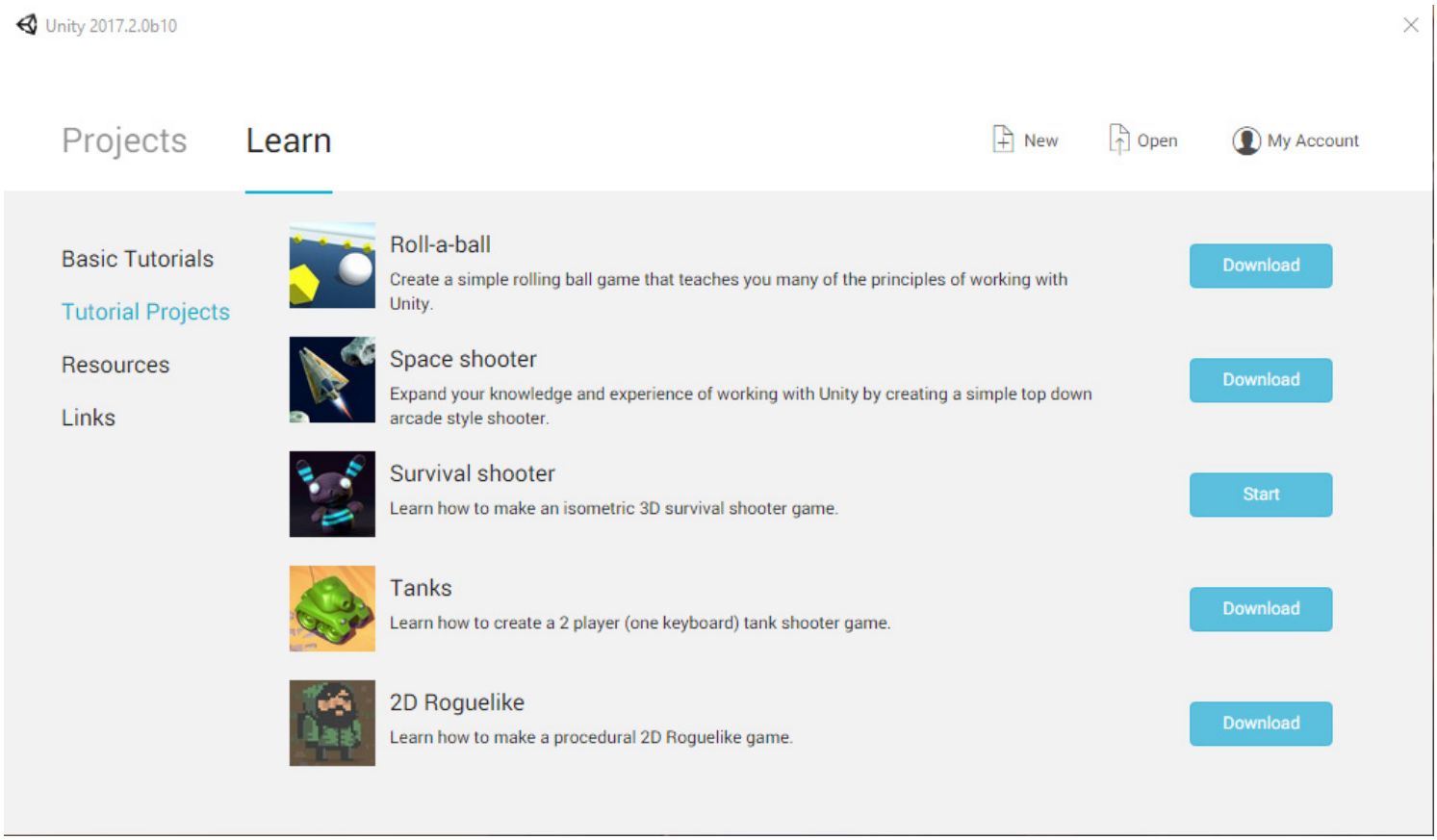
Basic Tutorials



The Home window with the Learn tab selected

The **Basic Tutorials** section contains interactive in-Editor tutorials. These tutorials take you through the very basics of interacting with Unity's interface and main development tools. To download each tutorial, click **Download**, then when the download is complete, click **Launch**.

Tutorial Projects



The **Tutorial Projects** section of the Learn tab

The **Tutorial Projects** section contains a list of tutorial Projects that you can import into Unity.

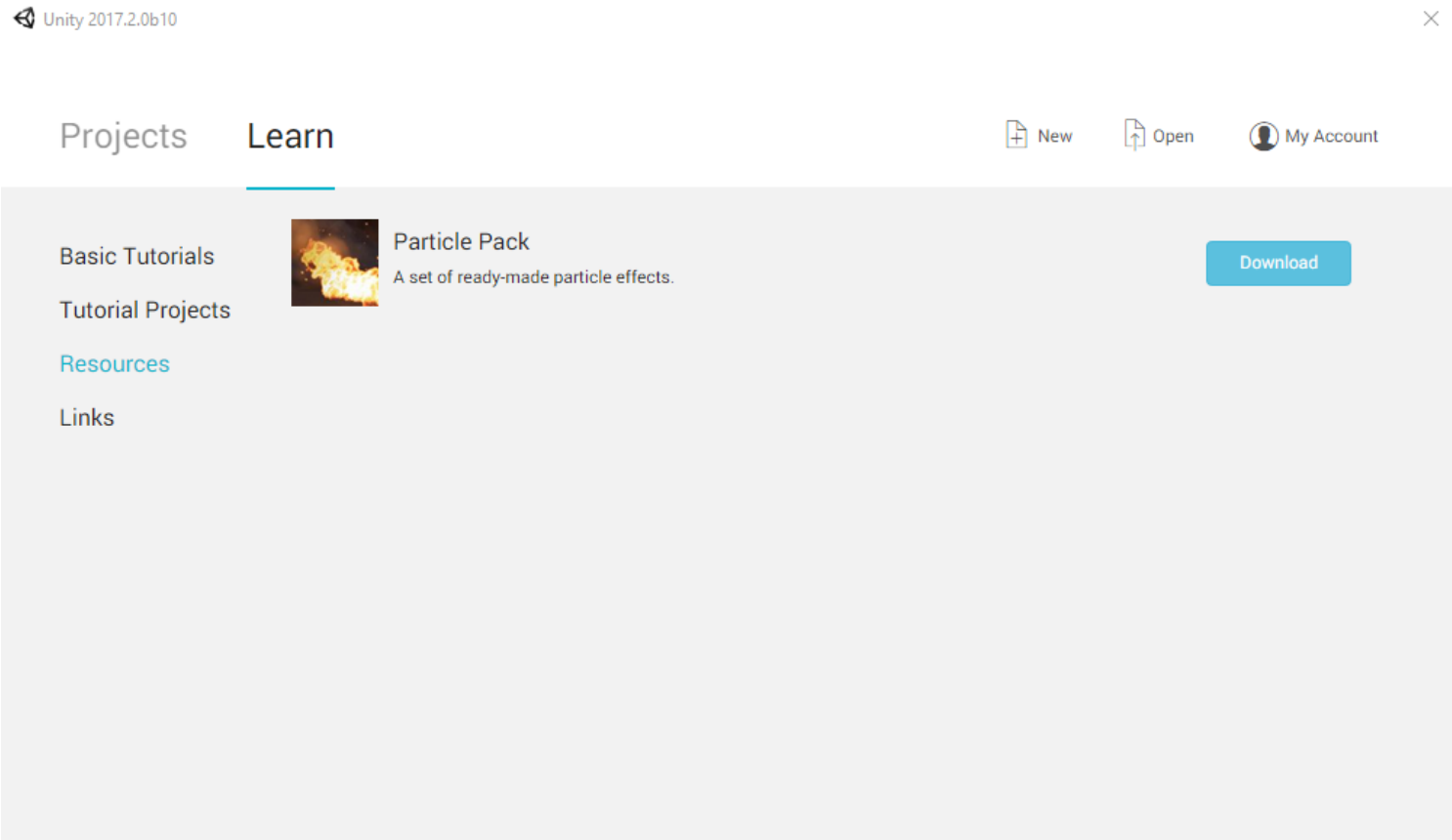
Each tutorial includes a full example Project, along with written and video guides, and all the resources you need to complete the Project yourself.

To download and import a tutorial Project into Unity, click the **Download** button to the right of the relevant Project. Unity then downloads all of the required **Assets** for the tutorial Project you have selected. When the download has finished, click the **Start** button to automatically create and open a new Project that contains all of the resources you need to follow the tutorial.

To access the video guides and full tutorials, click the **Read More** button to the right of the Project you want to know more about.

You can also follow a video tutorial for your chosen Project on the Unity Learn website.

Resources



The **Resources** section of the Learn tab

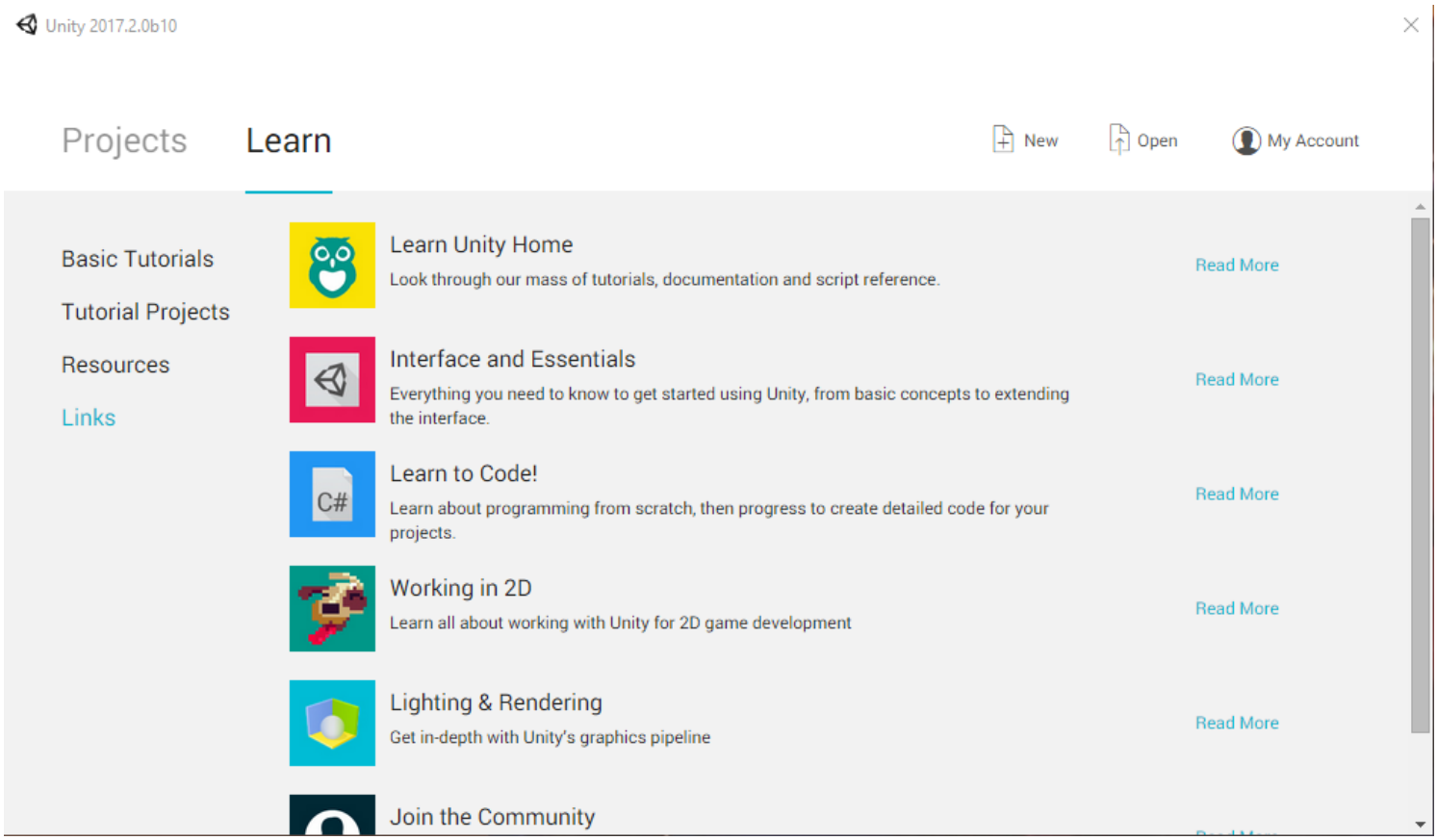
The **Resources** section contains links to Asset Packages that can be imported into your Project. Asset Packages contain resources such as 3D models, Particle effects, and pre-made **scripts** that you can use to quickly build Projects.

To download an Asset Package from the Learn tab, click the **Download** button to the right of the Asset Package you want to download. Unity then automatically downloads all of the Assets included in your chosen Asset Package.

To use these Assets in a Project, click the **Launch** button. Unity then automatically creates and opens a Project containing the downloaded Assets.

When downloading tutorials or resources, you can click the **X** button to the right of the progress bar to cancel the download.

Links



The **Links** section of the Learn tab

The **Links** section contains links to Unity Learn guides and tutorials, along with a link to the Unity Community homepage.

Click the **Read More** button to the right of the link description to open it in a new internet browser window.

2017–05–08 Page amended with limited editorial review

In-Editor Tutorials added in 2017.2

Opening existing Projects

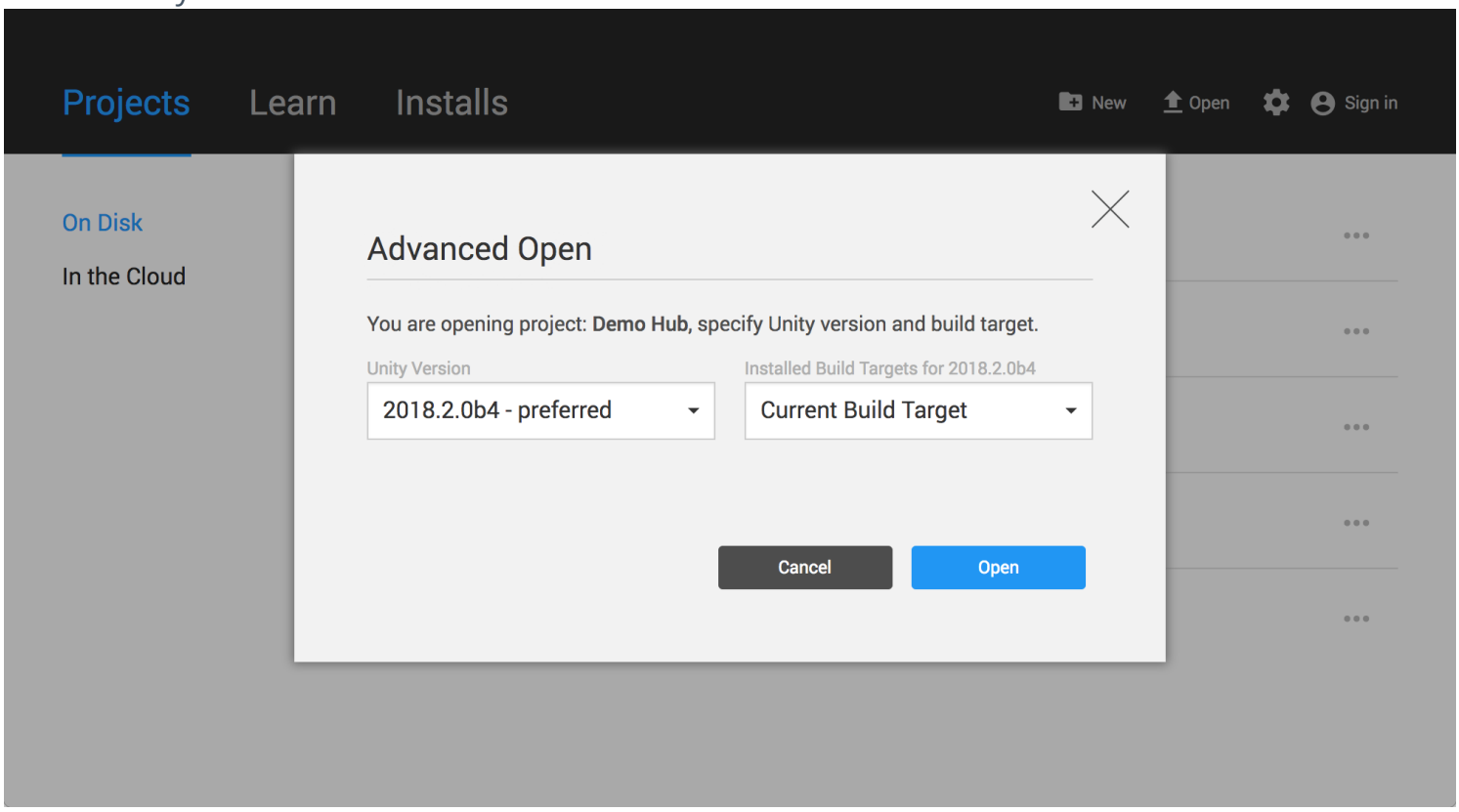
Leave feedback

This page details how to open an existing Project from both the Hub and the Unity Launcher.

Opening a Project from the Hub

You have several options when opening an existing Project from the Hub. You can:

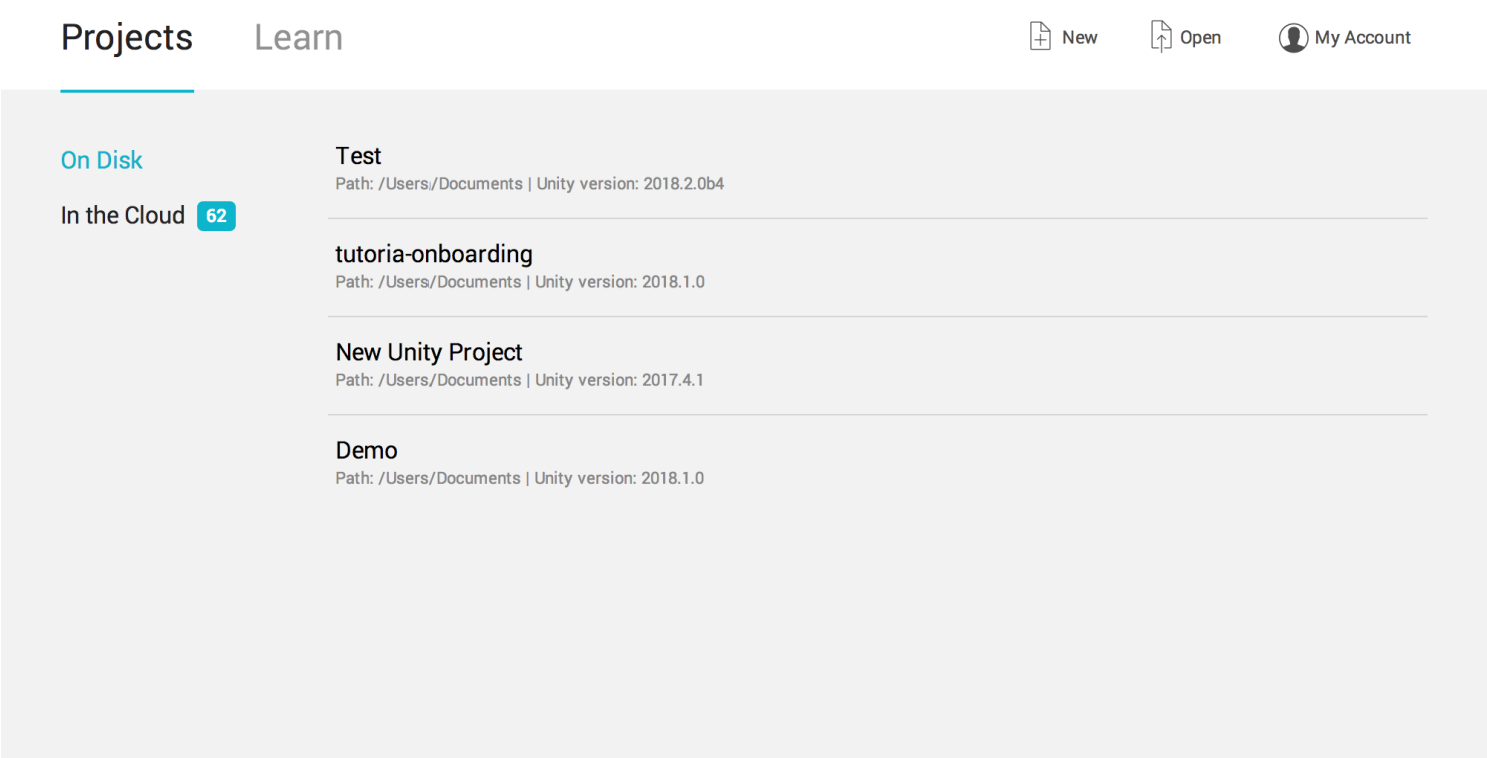
Click on the Project to open it using the assigned Editor version and the target platform.
Use the **Advanced Open** dialog to select a different Editor version or to specify a different target platform. To use **Advanced Open**, click the three dots to the right of the Project name and select the version you want to use.



The Hub also provides you with the means to open a Project with any installed version of the Editor. When you click **Open** to work with an existing Project, the Unity Hub attempts to open the Project with the corresponding Editor version for the Project. If the Hub can't find a matching Editor version for the Project, it displays a warning message and gives you the option to download the selected version, or open the Project with your preferred version.

Opening a Project from the Launcher

The Home Screen's **Project** tab lists any Project you have previously opened on this computer. Click on a Project in the list to open it.



A Project listed in the Projects tab

If the Editor is newly installed, or you haven't yet opened the Project you need in this installation of Unity, click **Open** to open your file browser and locate the Project folder. Note that a Unity Project is a collection of files and directories, rather than just one specific Unity Project file. To open a Project, you must select the main Project folder, rather than a specific file.

To view the Home Screen's **Projects** tab from inside the Unity Editor, go to **File** > **Open Project**.

2018–06–12 Page published with editorial review

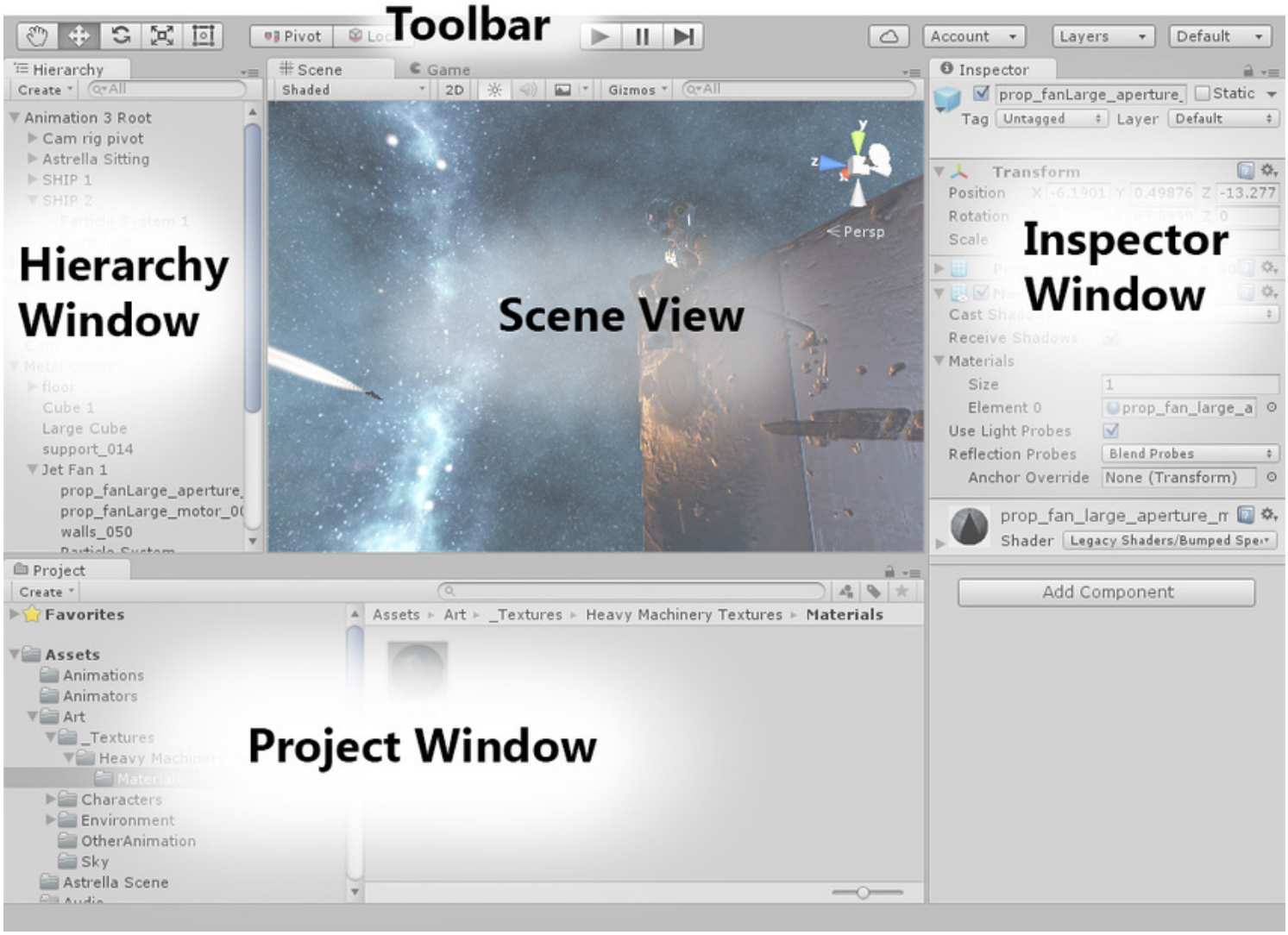
Learning the interface

Leave feedback

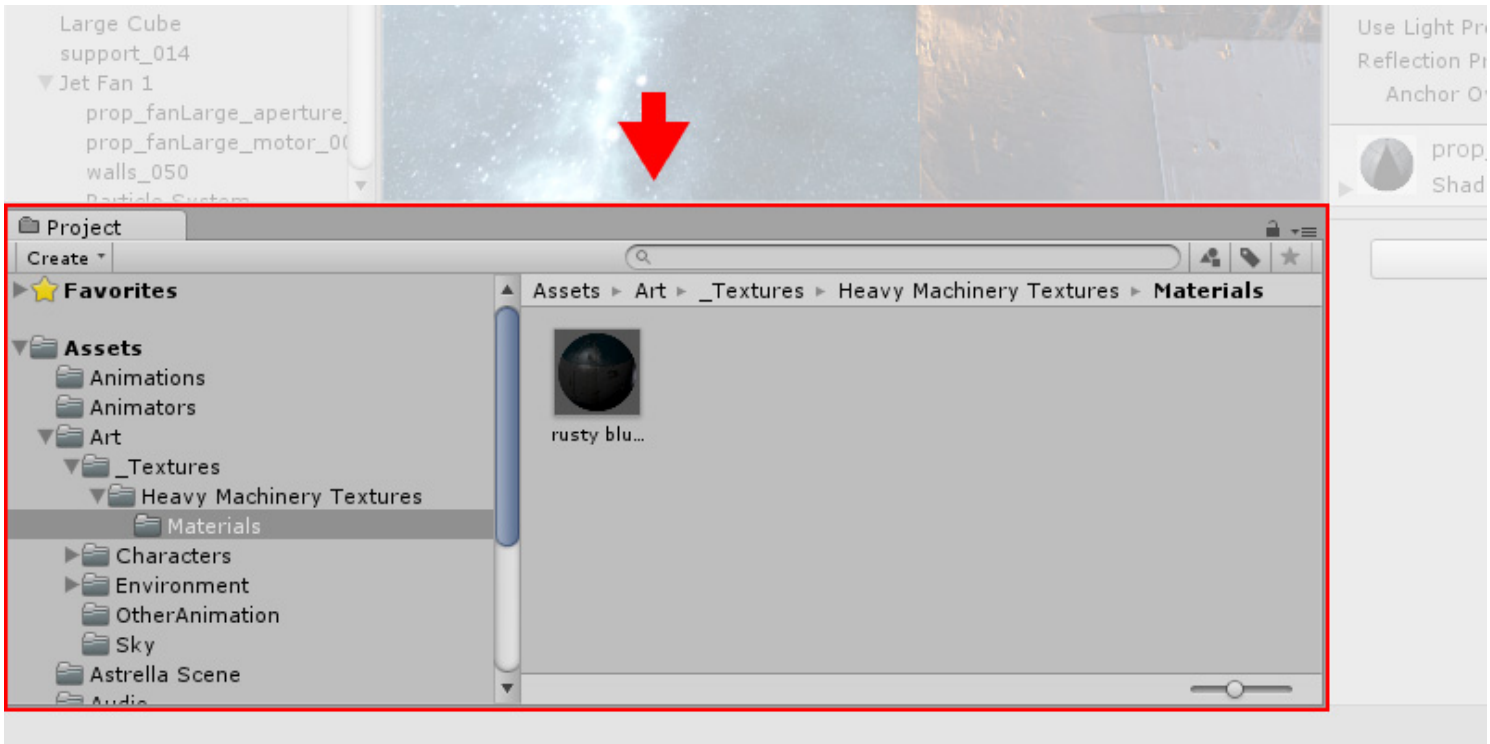
Take your time to look over the editor interface and familiarize yourself with it. The main editor window is made up of tabbed windows which can be rearranged, grouped, detached and docked.

This means the look of the editor can be different from one project to the next, and one developer to the next, depending on personal preference and what type of work you are doing.

The default arrangement of windows gives you practical access to the the most common windows. If you are not yet familiar with the different windows in Unity, you can identify them by the name in the tab. The most common and useful windows are shown in their default positions, below:

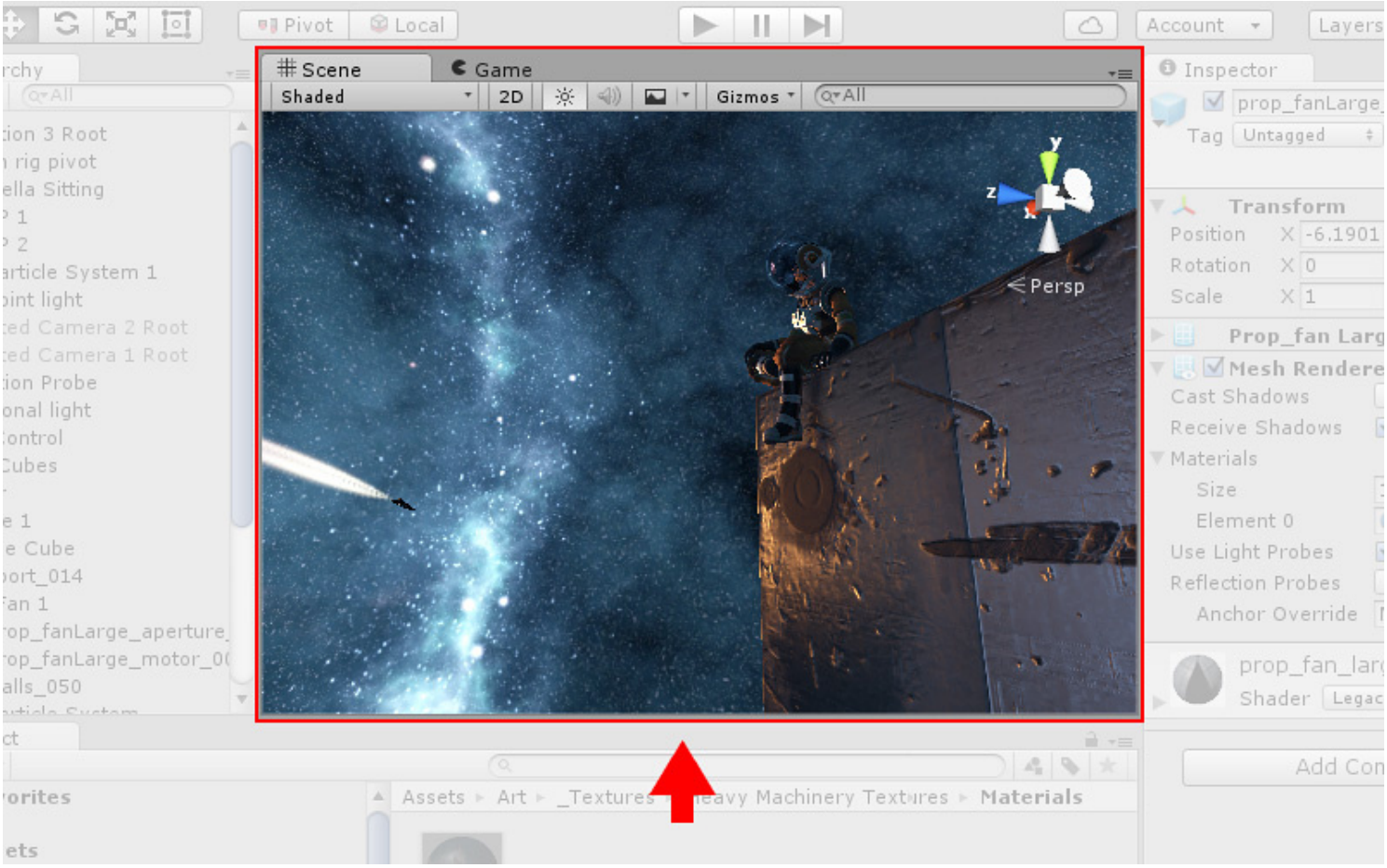


The Project Window



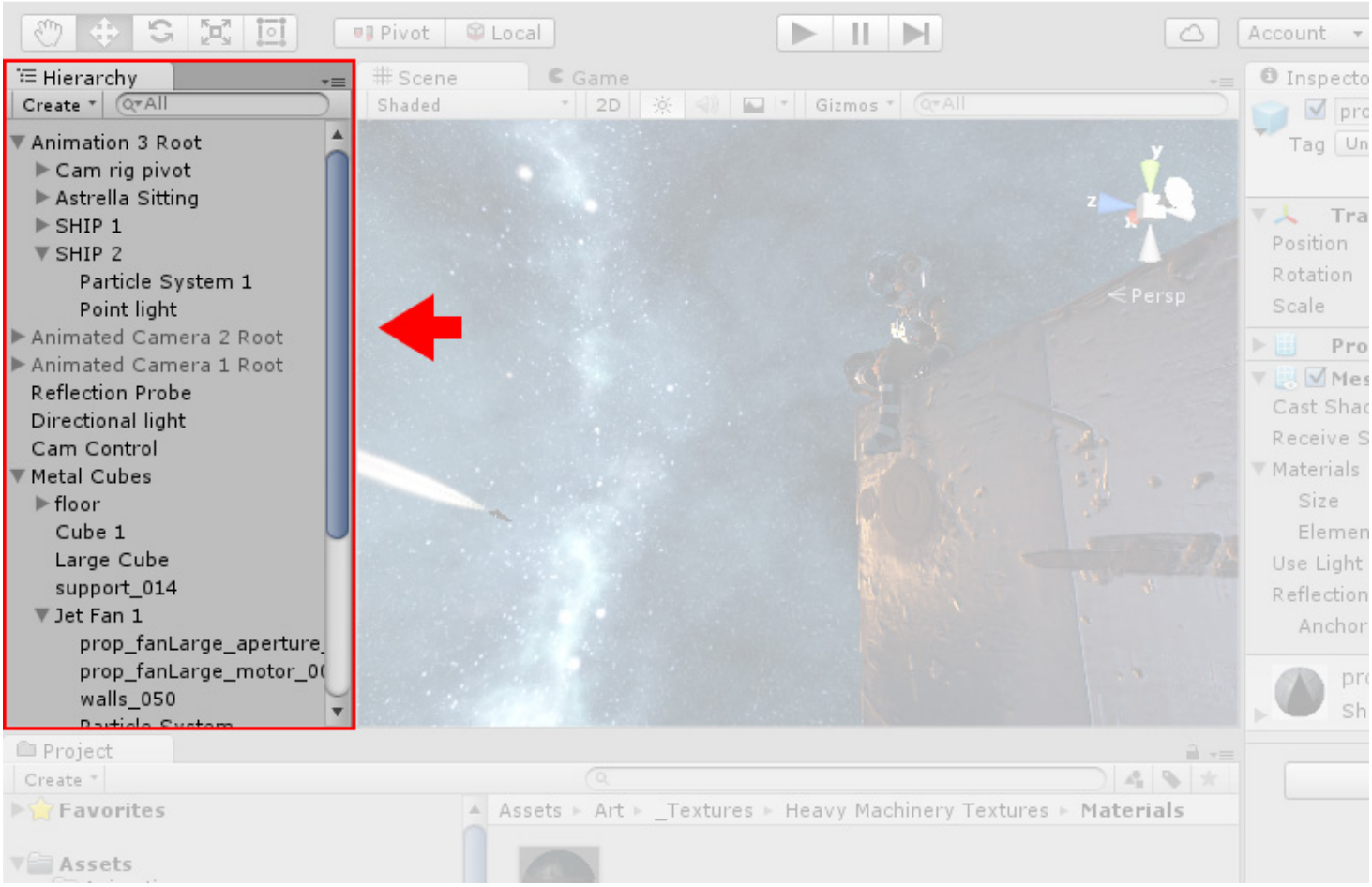
The Project Window displays your library of **assets** that are available to use in your project. When you import **assets** into your project, they appear here. Find out more about the Project Window.

The Scene View



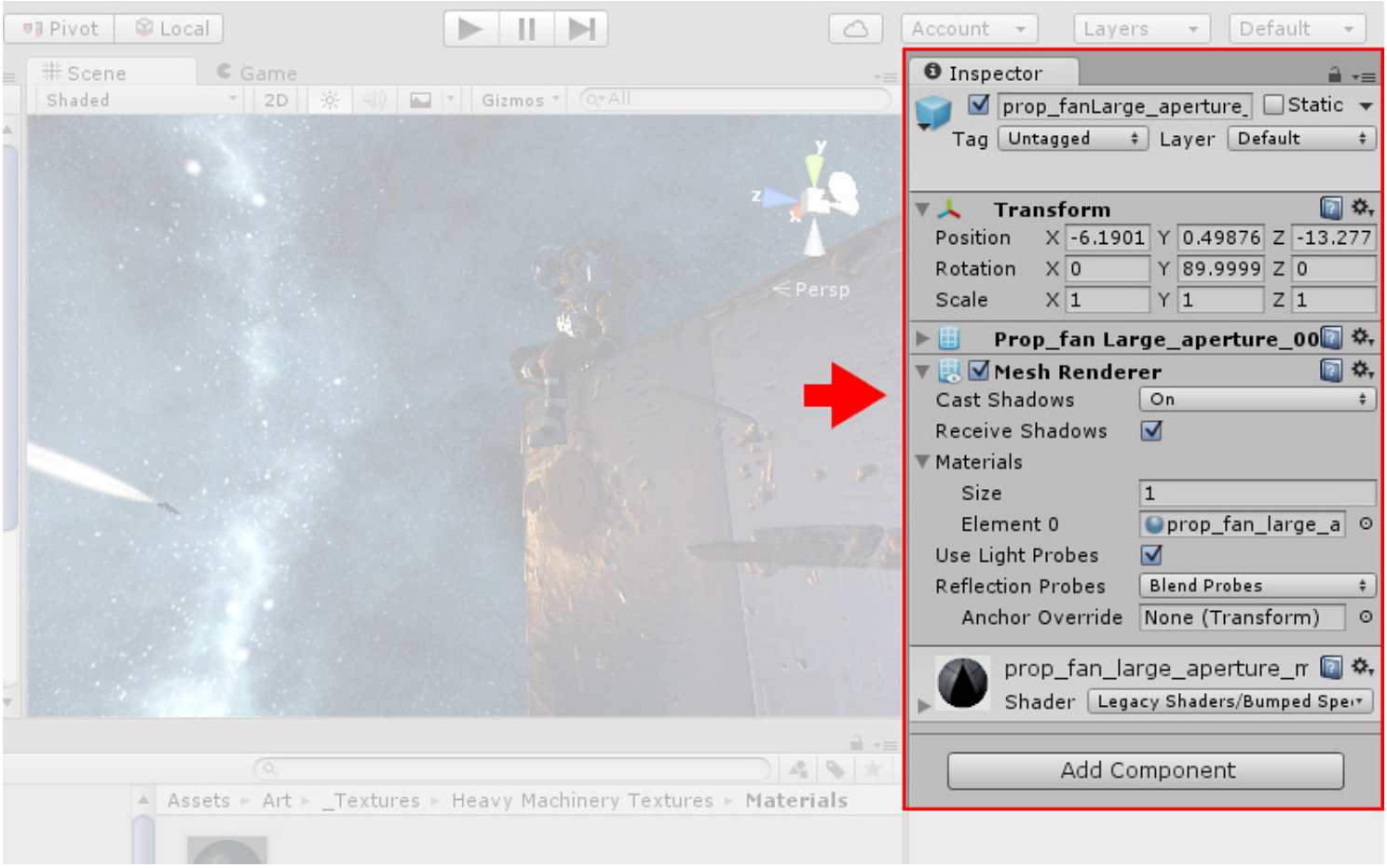
The Scene View allows you to visually navigate and edit your **scene**. The **scene** view can show a 3D or 2D perspective, depending on the type of project you are working on. Find out more about the Scene View and the Game View.

The Hierarchy Window



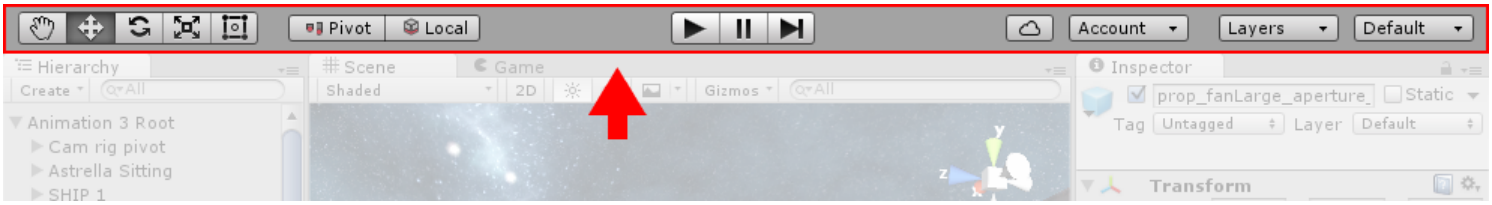
The Hierarchy Window is a hierarchical text representation of every object in the scene. Each item in the scene has an entry in the hierarchy, so the two windows are inherently linked. The hierarchy reveals the structure of how objects are attached to one another. Find out more about the Hierarchy Window.

The Inspector Window



The Inspector Window allows you to view and edit all the properties of the currently selected object. Because different types of objects have different sets of properties, the layout and contents of the **inspector** window will vary. Find out more about the Inspector Window.

The Toolbar



The Toolbar provides access to the most essential working features. On the left it contains the basic tools for manipulating the scene view and the objects within it. In the centre are the play, pause and step controls. The buttons to the right give you access to your Unity Cloud Services and your Unity Account, followed by a layer visibility menu, and finally the editor layout menu (which provides some alternate layouts for the editor windows, and allows you to save your own custom layouts).

The toolbar is not a window, and is the only part of the Unity interface that you can't rearrange.

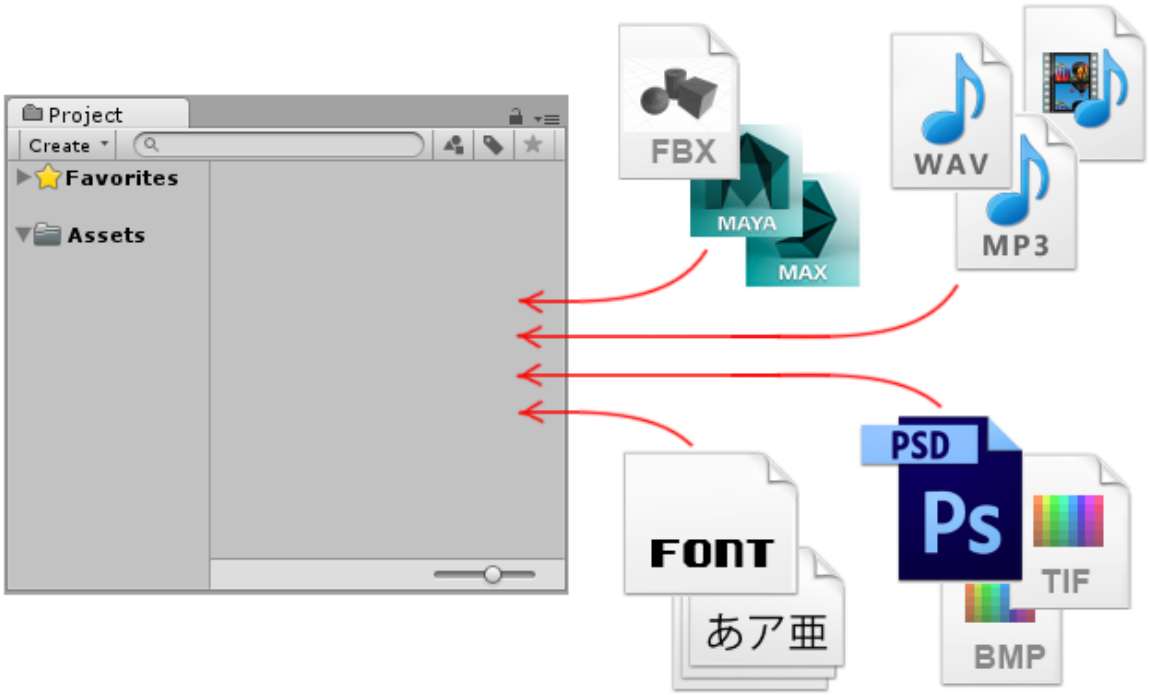
Find out more about the Toolbar.

Asset Workflow

Leave feedback

These steps will give you a general overview of the basic principles of working with **assets** in Unity.

An asset is representation of any item that can be used in your game or project. An asset may come from a file created outside of Unity, such as a 3D model, an audio file, an image, or any of the other types of file that Unity supports. There are also some asset types that can be created within Unity, such as an **Animator Controller**, an Audio Mixer or a **Render Texture**.



Some of the asset types that can be imported into Unity

Common types of Assets

Leave feedback

Image files

Unity supports most common image file types, such as BMP, TIF, TGA, JPG, and PSD. If you save your layered Photoshop (.psd) files into your **Assets** folder, Unity imports them as flattened images. You can find out more about importing images with alpha channels from photoshop, or importing your images as sprites

FBX and Model files

Since Unity supports the FBX file format, you can import data from any 3D modeling software that supports FBX. Unity also natively supports importing SketchUp files. For information on how to get the best results when exporting your FBX files from your 3D modeling software, see Optimizing FBX files.

Note: You can also save your 3D files from most common 3D modeling software in their native format (for example, .max, .blend, .mb, .ma). When Unity finds them in your Assets folder, it imports them by calling back to your 3D modeling software's FBX export plug-in. However, it is recommended to export them as FBX

Meshes and animations

Whichever 3D modeling software you are using, Unity imports the Meshes and animations from each file. For a list of 3D modeling software that Unity supports, see Model file formats.

Your **Mesh** file does not need to have an animation to be imported. If you do use animations, you can import all animations from a single file, or import separate files, each with a single animation. For more information about importing animations, see Model import workflows.

Audio files

If you save uncompressed audio files into your Assets folder, Unity imports them according to the **compression** settings specified. Read more about importing audio files.

Other Asset types

In all cases, your original source file is never modified by Unity, even though within Unity you can often choose between various ways to compress, modify, or otherwise process the Asset. The import process reads your source file, and creates a game-ready representation of your Asset internally, matching your chosen import settings. If you modify the import settings for an Asset, or make a change to the source file in the Asset folder, that causes Unity to re-import the Asset again to reflect your changes.

Note: *Importing native 3D formats requires that the 3D modeling software be installed on the same computer as Unity. This is because Unity uses the 3D modeling software's FBX Exporter plug-in to read the file. Alternatively, you can directly export as FBX from your application and save into the Projects folder.*

Standard Assets

Unity ships with multiple **Standard Assets**. These are collections of Assets that are widely used by most Unity customers. These include: 2D, **Cameras**, Characters, CrossPlatformInput, Effects, Environment, ParticleSystems, Prototyping, Utility, Vehicles.

Unity transfers **Standard Assets** in and out of Projects using **Asset packages**.

Note: If you chose not to install Standard Assets when you installed Unity, you can download them from the Asset Store.

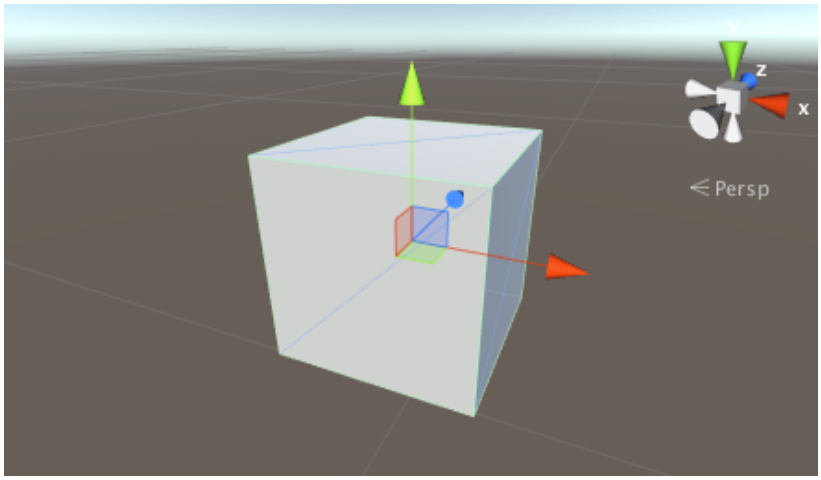
2018–04–25 Page amended with limited editorial review

Primitive and placeholder objects

Leave feedback

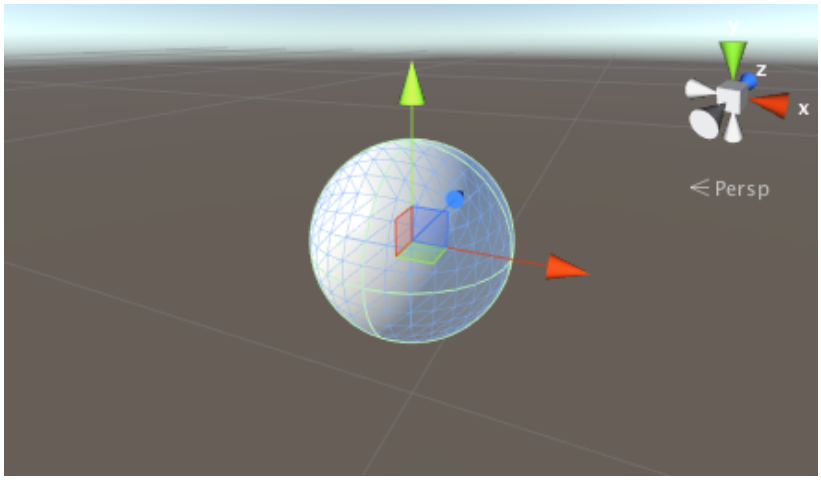
Unity can work with 3D models of any shape that can be created with modeling software. However, there are also a number of primitive object types that can be created directly within Unity, namely the **Cube**, **Sphere**, **Capsule**, **Cylinder**, **Plane** and **Quad**. These objects are often useful in their own right (a plane is commonly used as a flat ground surface, for example) but they also offer a quick way to create placeholders and prototypes for testing purposes. Any of the primitives can be added to the scene using the appropriate item on the **GameObject > 3D Object** menu.

Cube



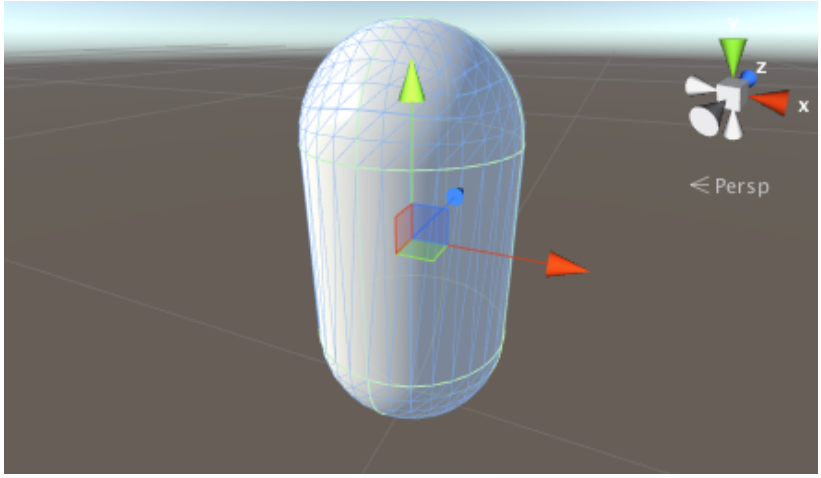
This is a simple cube with sides one unit long, textured so that the image is repeated on each of the six faces. As it stands, a cube isn't really a very common object in most games but when scaled, it is very useful for walls, posts, boxes, steps and other similar items. It is also a handy placeholder object for programmers to use during development when a finished model is not yet available. For example, a car body can be crudely modelled using an elongated box of roughly the right dimensions. Although this is not suitable for the finished game, it is fine as a simple representative object for testing the car's control code. Since a cube's edges are one unit in length, you can check the proportions of a **mesh** imported into the **scene** by adding a cube close by and comparing the sizes.

Sphere



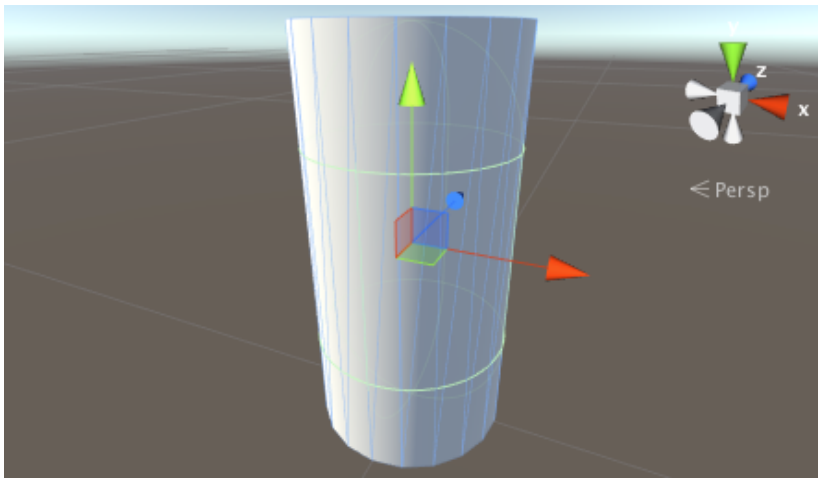
This is a sphere of unit diameter (ie, 0.5 unit radius), textured so that the entire image wraps around once with the top and bottom "pinched" at the poles. Spheres are obviously useful for representing balls, planets and projectiles but a semi-transparent sphere can also make a nice GUI device for representing the radius of an effect.

Capsule



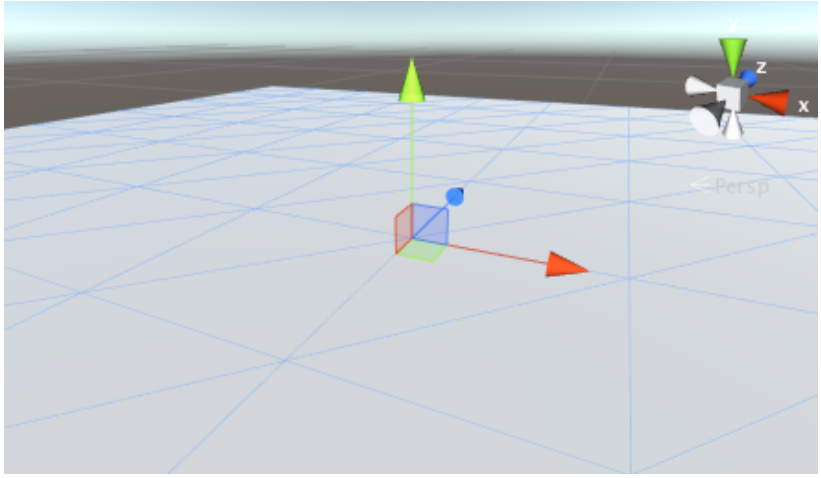
A capsule is a cylinder with hemispherical caps at the ends. The object is one unit in diameter and two units high (the body is one unit and the two caps are half a unit each). It is textured so that the image wraps around exactly once, pinched at each hemisphere's apex. While there aren't many real world objects with this shape, the capsule is a useful placeholder for prototyping. In particular, the physics of a rounded object are sometimes better than those of a box for certain tasks.

Cylinder



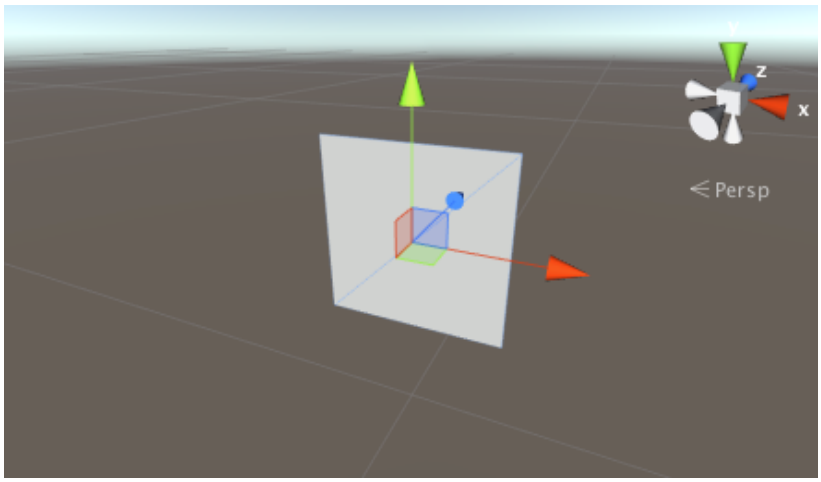
This is a simple cylinder which is two units high and one unit in diameter, textured so that the image wraps once around the tube shape of the body but also appears separately in the two flat, circular ends. Cylinders are very handy for creating posts, rods and wheels but you should note that the shape of the **collider** is actually a capsule (there is no primitive cylinder collider in Unity). You should create a mesh of the appropriate shape in a modeling program and attach a **mesh collider** if you need an accurate cylindrical collider for physics purposes.

Plane



This is a flat square with edges ten units long oriented in the XZ plane of the local coordinate space. It is textured so that the whole image appears exactly once within the square. A plane is useful for most kinds of flat surface, such as floors and walls. A surface is also needed sometimes for showing images or movies in GUI and special effects. Although a plane *can* be used for things like this, the simpler **quad** primitive is often a more natural fit to the task.

Quad



The quad primitive resembles the plane but its edges are only one unit long and the surface is oriented in the XY plane of the local coordinate space. Also, a quad is divided into just two triangles whereas the plane contains two hundred. A quad is useful in cases where a scene object must be used simply as a display screen for an image or movie. Simple GUI and information displays can be implemented with quads, as can particles, **sprites** and “impostor” images that substitute for solid objects viewed at a distance.

2018–04–25 Page amended with limited editorial review

Asset packages

Leave feedback

Unity uses two types of packages:

Asset packages , available on the Unity **Asset Store**, which allow you to share and re-use Unity Projects and collections of Assets.
Unity packages, available through the Package Manager window. You can import a wide range of Assets, including plug-ins directly into Unity with this type of package.
This section provides information about using Asset packages in Unity.

Asset packages

Unity **Standard Assets** and items on the Unity **Asset Store** are supplied in packages, which are collections of files and data from Unity Projects, or elements of Projects, which are compressed and stored in one file, similar to zip files. Like zip files, a package maintains its original directory structure when it is unpacked, as well as meta-data about Assets (such as import settings and links to other Assets).

In Unity, the menu option **Export Package** compresses and stores the collection, while **Import Package** unpacks the collection into your currently open Unity Project.

This page contains information on:

Importing packages (both **Standard Asset** packages and custom packages)
Exporting packages (both new and updated)

Importing Asset packages

You can import **Standard Asset Packages**, which are Asset collections pre-made and supplied with Unity, and **Custom Packages**, which are made by people using Unity.

Choose **Assets** > **Import Package** to import both types of Asset packages.

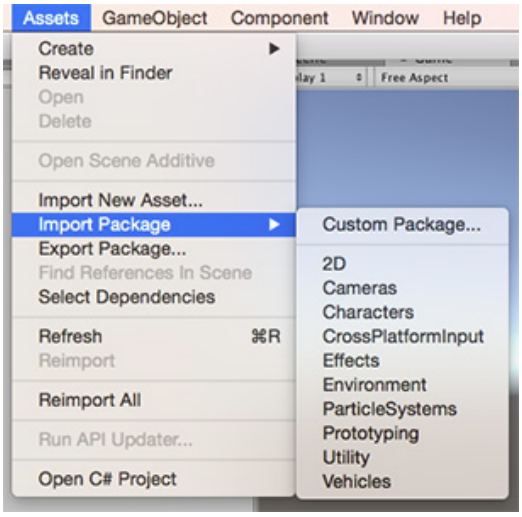


Fig 1: Asset > Import Package menu

Importing Standard Assets

Unity **'Standard Assets'** consist of several different packages: **2D,** Cameras**, Characters, CrossPlatformInput, Effects, Environment, ParticleSystems, Prototyping, Utility, Vehicles**.

To import a new **Standard Asset** package:

Open the Project you want to import Assets into.

Choose **Assets** > **Import Package** and then select the name of the package you want to import from the list. The **Import Unity Package** dialog box appears with all the items in the package pre-checked, ready to install. (See *Fig 2: New install Import Unity Package*

Dialog Box.)

Select **Import** and Unity puts the contents of the package into a **Standard Asset** folder, which you can access from your **Project View**.

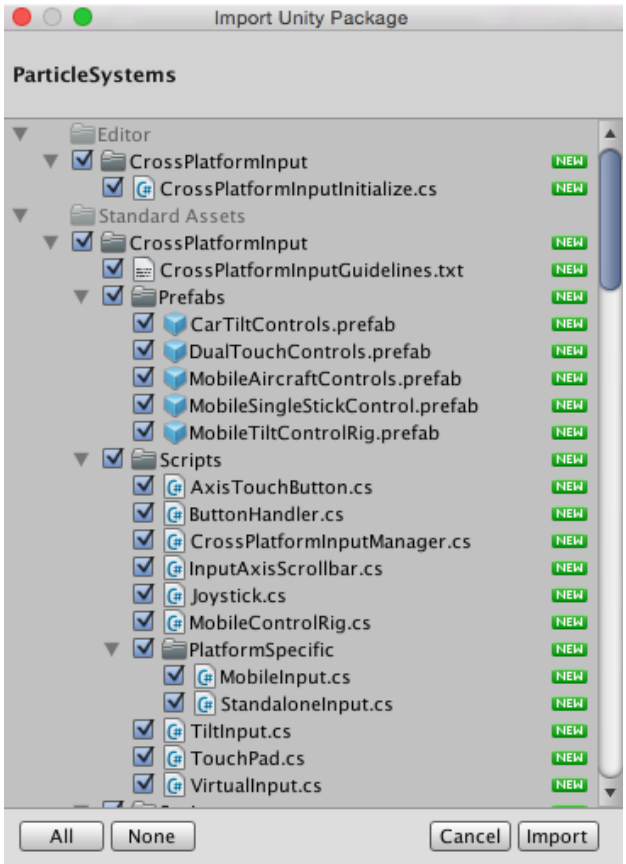


Fig 2: New install Import Unity Package dialog box

Importing custom Asset packages

You can import custom packages which have been exported from your own Projects or from Projects made by other Unity users.

To import a new custom package:

Open the Project you want to import Assets into.

Choose **Assets** > **Import Package** > **Custom Package...** to bring up up File Explorer (Windows) or Finder (Mac).

Select the package you want from Explorer or Finder, and the **Import Unity Package** dialog box displays, with all the items in the package pre-checked, ready to install. (See *Fig 4: New install Import Unity Package dialog box*.)

Select **Import** and Unity puts the contents of the package into the **Assets** folder, which you can access from your **Project View**.

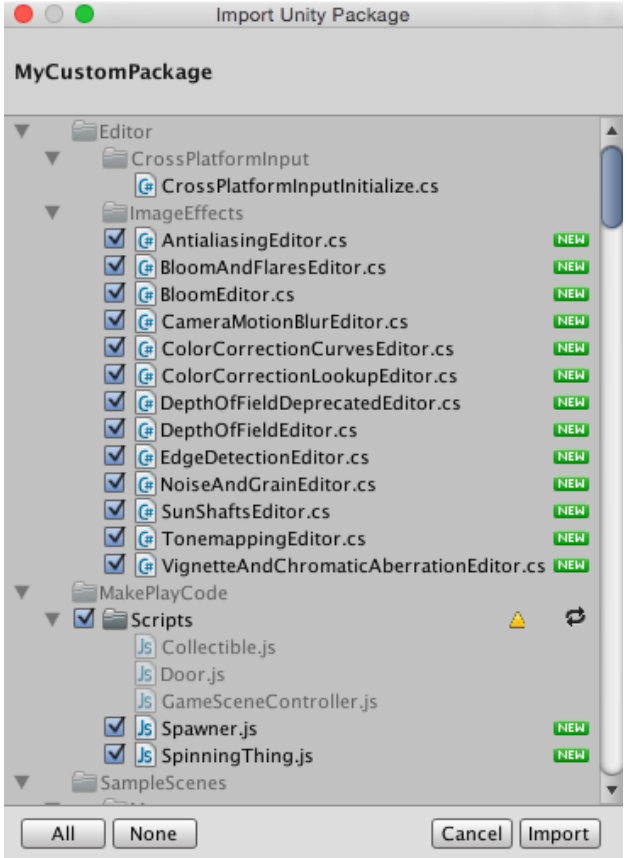


Fig 3: New install Import Unity Package dialog box

Upgrading Standard Assets

Standard Assets do not upgrade automatically when you upgrade the Editor.

When you create a new Project in Unity, you can choose to include **Standard Assets** collections in your Project. Unity copies the Assets you choose to include from the Unity install folder into your new Project folder. This means that if you upgrade your Unity Editor to a newer version, the **Standard Assets** you have already imported into your Project do not upgrade: you have to manually upgrade them.

Hint: A newer version of a **Standard Asset** might behave differently in your existing installation (for performance or quality reasons, for example). A newer version might make your Project look or behave differently and you may need to re-tweak its parameters. Check the package contents and Unity's release notes before you decide to re-install.

Exporting packages

Use **Export Package** to create your own **Custom Package**.

Open the Project you want to export Assets from.

Choose **Assets > Export Package...** from the menu to bring up the **Exporting Package** dialog box. (See *Fig 4: Exporting Package dialog box*.)

In the dialog box, select the Assets you want to include in the package by clicking on the boxes so they are checked.

Leave the **include dependencies** box checked to auto-select any Assets used by the ones you have selected.

Click on **Export** to bring up File Explorer (Windows) or Finder (Mac) and choose where you want to store your package file.

Name and save the package anywhere you like.

Hint: When exporting a package Unity can export all dependencies as well. So, for example, if you select a **Scene** and export a package with all dependencies, then all models, Textures and other Assets that appear in the scene will be exported as well. This can be a quick way of exporting a bunch of Assets without manually locating them all.

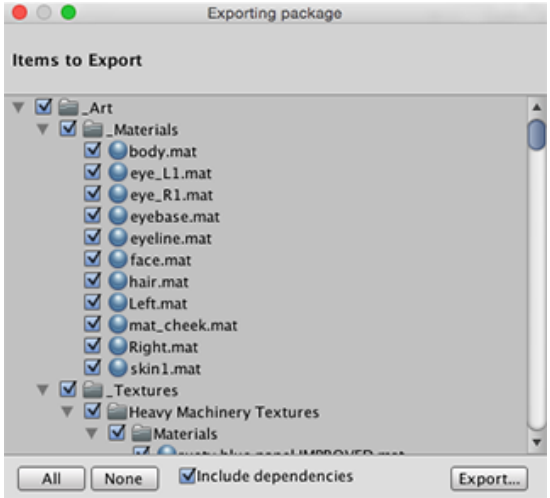


Fig 4: Exporting Package dialog box

Updating packages

Sometimes you may want to change the contents of a package and create a newer, updated version of your Asset package. To do this:

Select the Asset files you want in your package (select both the unchanged ones and the new ones).

Export the files as described above in ***Export Package*** , above.

Note: You can re-name an updated package and Unity will recognise it as an update, so you can use incremental naming, for example: MyAssetPackageVer1, MyAssetPackageVer2.

Hint: It is not good practice to remove files from packages and then replace them with the same name: Unity will recognize them as different and possibly conflicting files and so display a warning symbol when they are imported. If you have removed a file and then decide to replace it, it is better to give it a different but related name to the original.

2018–04–25 Page amended with limited editorial review

Using the Asset Store

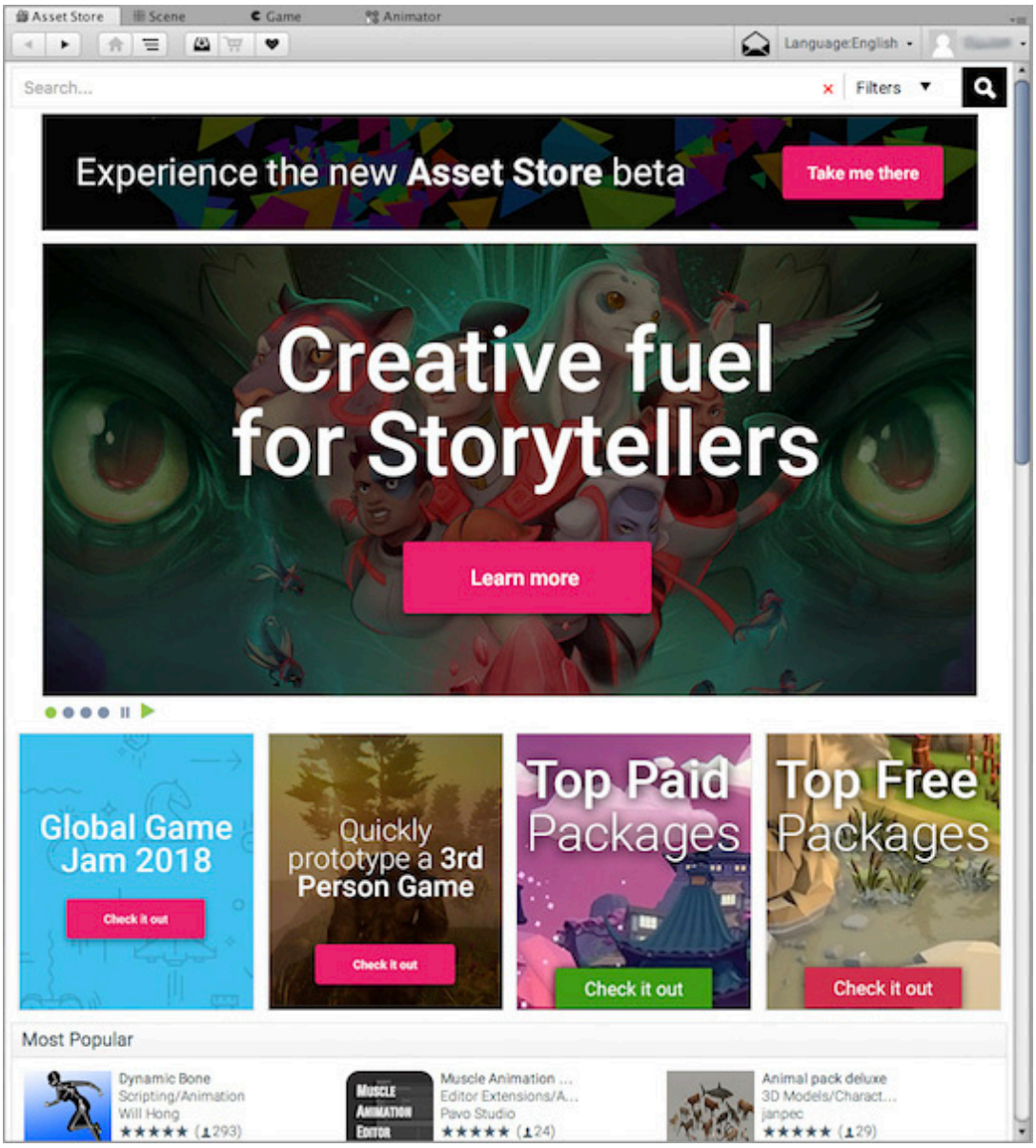
Leave feedback

The Unity **Asset Store** is home to a growing library of free and commercial **Assets** created both by Unity Technologies and also members of the community. A wide variety of **Assets** is available, covering everything from Textures, Models and animations to whole Project examples, tutorials and Editor extensions. You can access the **Assets** from a simple interface built into the Unity Editor which allows you to download and import Assets directly into your Project.

Unity users can become publishers on Asset Store, and sell the content they have created. To find out more, see Asset Store information on Asset Store Publishing.

Asset Store access and navigation

To open the Asset Store window, select **Window** > **General** > **Asset Store** from the main menu in Unity. During your first visit, you can create a free user account which allows you to log into the Store on future visits and keep track of previous purchases and downloads.

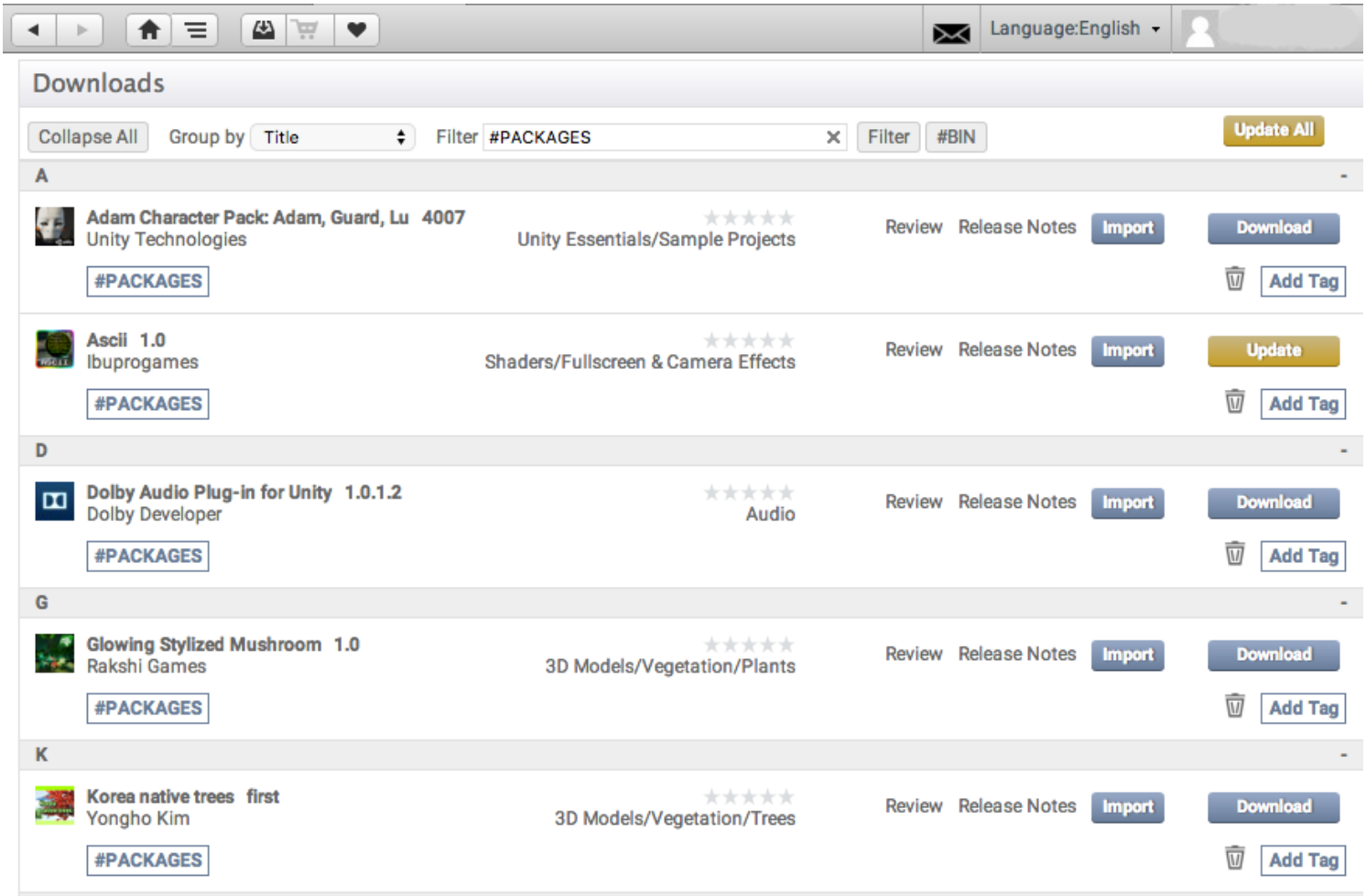


The Asset Store front page.

The Store provides a browser-like interface which allows you to navigate either by free text search or by browsing packages and categories. Standard navigation buttons appear to the left of the main **toolbar** which you can use to navigate through your browsing history:

The Download Manager and Shopping Cart buttons appear to the right of the navigation buttons which allow you to open the Download Manager or view the current contents of your shopping cart:

The Download Manager allows you to view the packages you have already bought as well as find and install any updates. You can also use it to view the Standard **Asset packages** supplied with Unity and add them to your Project.



The Download Manager

Location of downloaded Asset files

You rarely need to access the files downloaded from the Asset Store directly. However, if you do need to, you can find them in the following paths:

macOS: `~/Library/Unity/Asset Store`
Windows: `C:\Users\accountName\AppData\Roaming\Unity\Asset Store`
These folders contain subfolders that correspond to particular Asset Store vendors. The actual Asset files are contained inside the subfolders using a structure defined by the package publisher.

2018–05–11 Page amended with limited editorial review

2018–04–25 Page amended with limited editorial review

2018–01–31 Page amended with limited editorial review

The Main Windows

Leave feedback

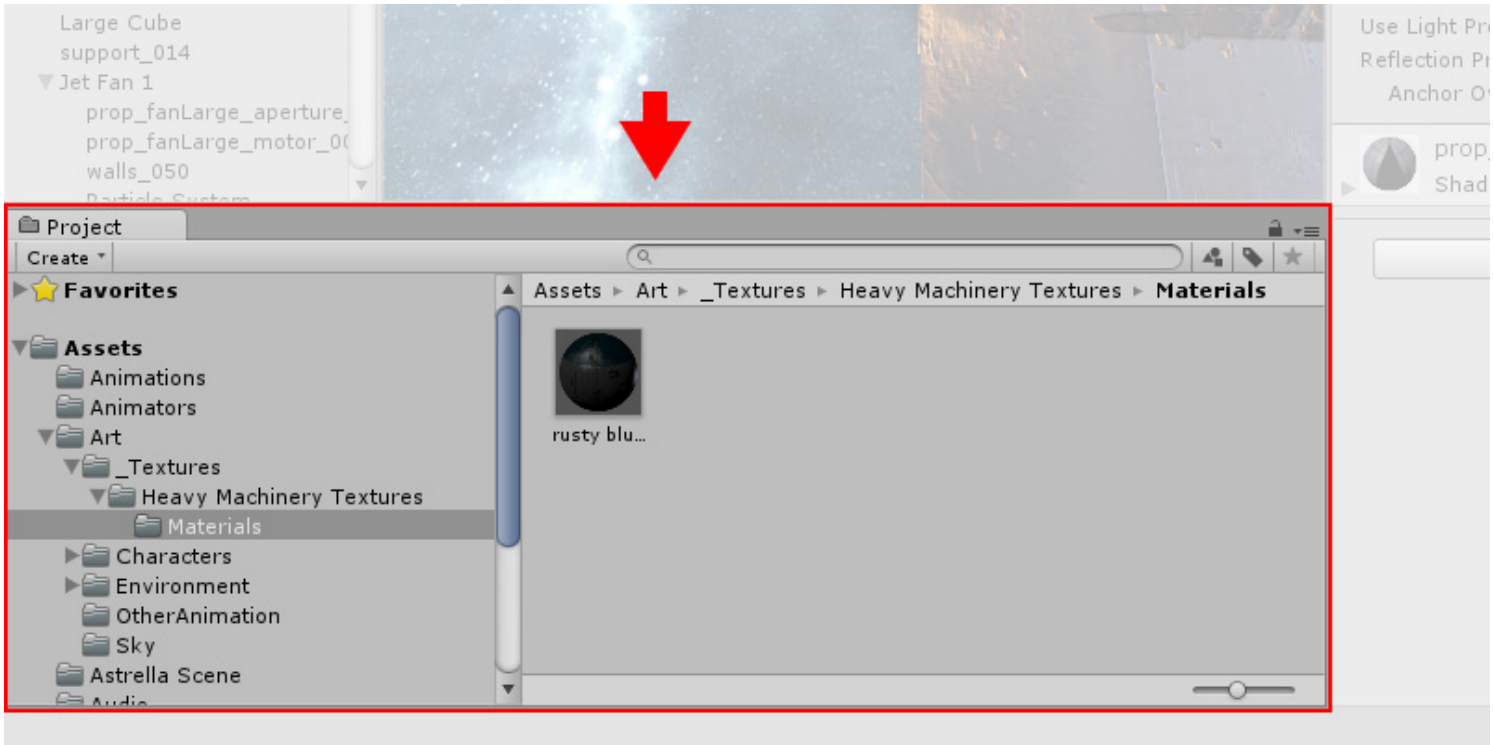
This section provides a detailed tour of the most common editor windows, and how to make full use of them.

The Project Window
The Scene View
The Hierarchy Window
The Inspector Window
The Toolbar
The Game View

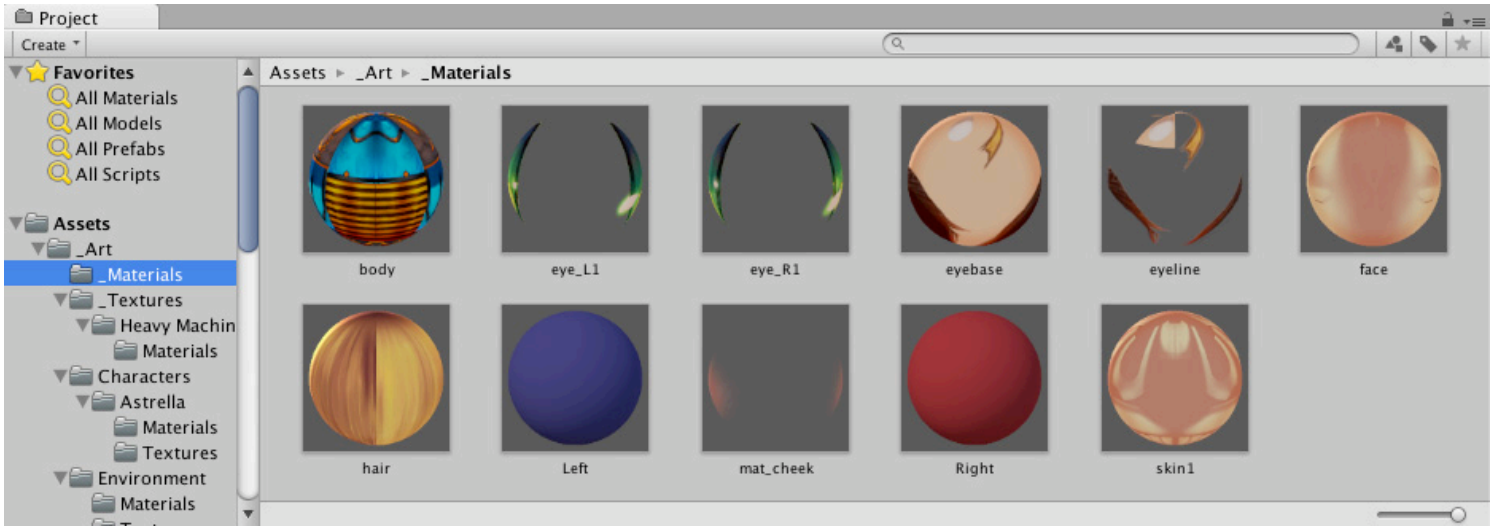
See the Knowledge Base Editor section for troubleshooting, tips and tricks.

The Project Window

Leave feedback



In this view, you can access and manage the **assets** that belong to your project.



The left panel of the browser shows the folder structure of the project as a hierarchical list. When a folder is selected from the list by clicking, its contents will be shown in the panel to the right. You can click the small triangle to expand or collapse the folder, displaying any nested folders it contains. Hold down Alt while you click to expand or collapse any nested folders recursively.

The individual assets are shown in the right hand panel as icons that indicate their type (script, material, sub-folder, etc). The icons can be resized using the slider at the bottom of the panel; they will be replaced by a hierarchical list view if the slider is moved to the extreme left. The space to the left of the slider shows the currently selected item, including a full path to the item if a search is being performed.

Above the project structure list is a **Favorites** section where you can keep frequently-used items for easy access. You can drag items from the project structure list to the Favourites and also save search queries there (see **Searching** below).

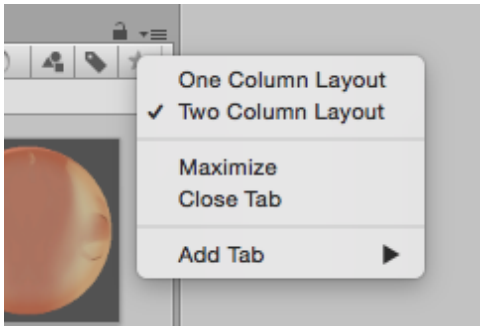
Just above the panel is a “breadcrumb trail” that shows the path to the folder currently being viewed. The separate elements of the trail can be clicked for easy navigation around the folder hierarchy. When searching, this bar changes to show the area being searched (the root Assets folder, the selected folder or the Asset Store) along with a count of free and paid assets available in the store, separated by a slash. There is an option in the *General* section of Unity’s Preferences window to disable the display of Asset Store hit counts if they are not required.

The ‘breadcrumb trail’ shows the path to the folder you are currently viewing

Along the top edge of the window is the browser’s **toolbar**.

Located at the left side of the toolbar, the **Create** menu lets you add new assets and sub-folders to the current folder. To its right are a set of tools to allow you to search the assets in your project.

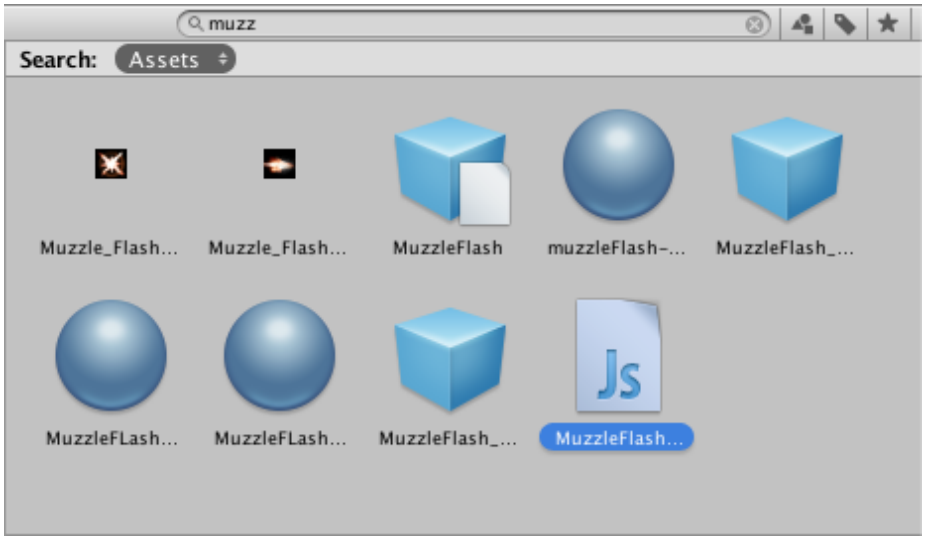
The Window menu provides the option of switching to a one-column version of the **project view**, essentially just the hierarchical structure list without the icon view. The lock icon next to the menu enables you to “freeze” the current contents of the view (ie, stop them being changed by events elsewhere) in a similar manner to the inspector lock.



In the top right corner, select the dropdown menu to change the view layout, and click on the lock icon to freeze the view

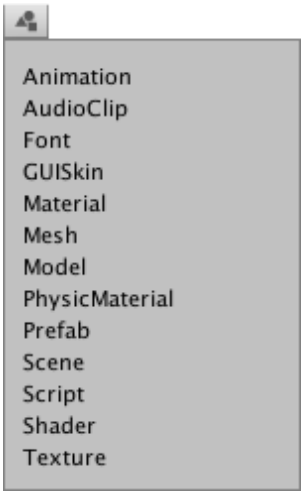
Searching

The browser has a powerful search facility that is especially useful for locating assets in large or unfamiliar projects. The basic search will filter assets according to the text typed in the search box

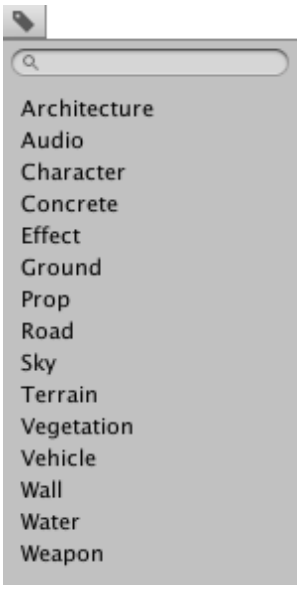


If you type more than one search term then the search is narrowed, so if you type *coastal scene* it will only find assets with both “coastal” and “**scene**” in the name (ie, terms are ANDed together).

To the right of the search bar are three buttons. The first allows you to further filter the assets found by the search according to their type.



Continuing to the right, the next button filters assets according to their Label (labels can be set for an asset in the Inspector). Since the number of labels can potentially be very large, the label menu has its own mini-search filter box.



Note that the filters work by adding an extra term in the search text. A term beginning with “t:” filters by the specified asset type, while “l:” filters by label. You can type these terms directly into the search box rather than use the menu if you know what you are looking for. You can search for more than one type or label at once. Adding several types will expand the search to include all specified types (ie, types will be ORed together). Adding multiple labels will narrow the search to items that have all the specified labels (ie, labels are ANDed).

The rightmost button saves the search by adding an item to the Favourites section of the asset list.

Searching the Asset Store

The Project Browser's search can also be applied to assets available from the Unity **Asset Store**. If you choose **Asset Store** from the menu in the breadcrumb bar, all free and paid items from the store that match your query will be displayed. Searching by type and label works the same as for a Unity project. The search query words will be checked against the asset name first and then the package name, package label and package description in that order (so an item whose name contains the search terms will be ranked higher than one with the same terms in its package description).

If you select an item from the list, its details will be displayed in the inspector along with the option to purchase and/or download it. Some asset types have previews available in this section so you can, for example, rotate a 3D model before buying. The inspector also gives the option of viewing the asset in the usual Asset Store window to see additional details.

Shortcuts

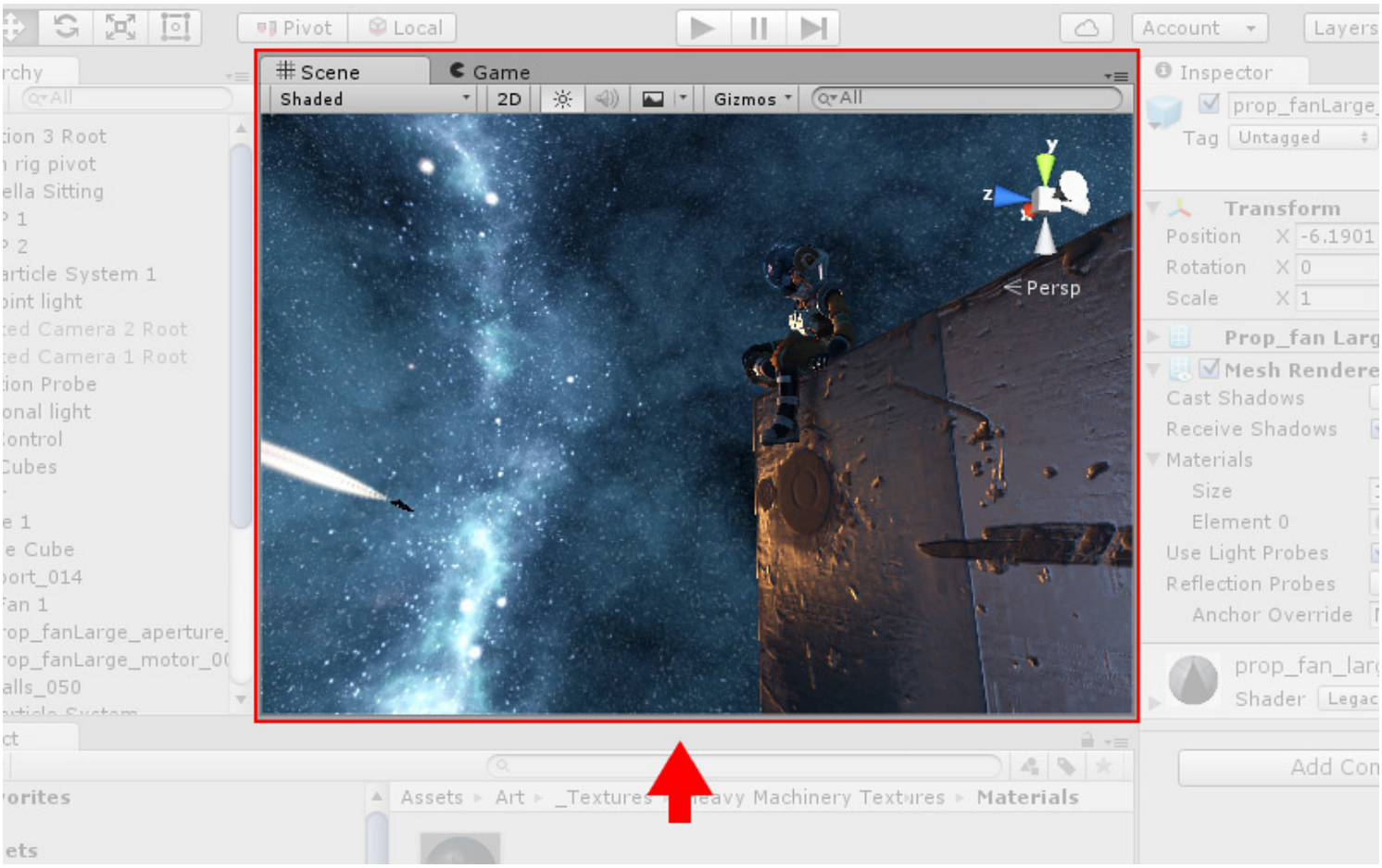
The following keyboard shortcuts are available when the browser view has focus. Note that some of them only work when the view is using the two-column layout (you can switch between the one- and two-column layouts using the panel menu in the very top right corner).

F	Frame selected (ie, show the selected asset in its containing folder)
Tab	Shift focus between first column and second column (Two columns)
Ctrl/Cmd + F	Focus search field
Ctrl/Cmd + A	Select all visible items in list
Ctrl/Cmd + D	Duplicate selected assets
Delete	Delete with dialog (Win)
Delete + Shift	Delete without dialog (Win)
Delete + Cmd	Delete without dialog (OSX)
Enter	Begin rename selected (OSX)
Cmd + down arrow	Open selected assets (OSX)
Cmd + up arrow	Jump to parent folder (OSX, Two columns)
F2	Begin rename selected (Win)
Enter	Open selected assets (Win)

Backspace	Jump to parent folder (Win, Two columns)
Right arrow	Expand selected item (tree views and search results). If the item is already expanded, this will select its first child item.
Left arrow	Collapse selected item (tree views and search results). If the item is already collapsed, this will select its parent item.
Alt + right arrow	Expand item when showing assets as previews
Alt + left arrow	Collapse item when showing assets as previews

The Scene View

Leave feedback



The **Scene View** is your interactive view into the world you are creating. You will use the **Scene** View to select and position scenery, characters, **cameras**, lights, and all other types of **Game Object**. Being able to Select, manipulate and modify objects in the **Scene** View are some of the first skills you must learn to begin working in Unity.

Scene View navigation

Leave feedback

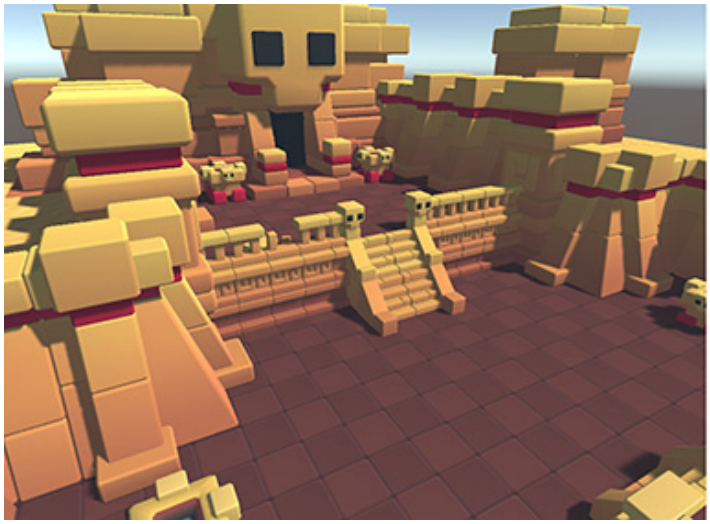
The **Scene View** has a set of navigation controls to help you move around quickly and efficiently.

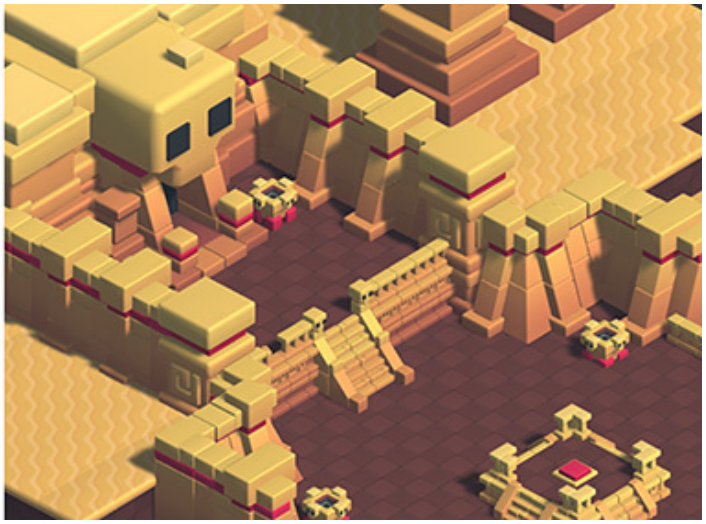
Scene Gizmo

The **Scene Gizmo** is in the upper-right corner of the **Scene** View. This displays the **Scene** View **Camera**'s current orientation, and allows you to quickly modify the viewing angle and projection mode.

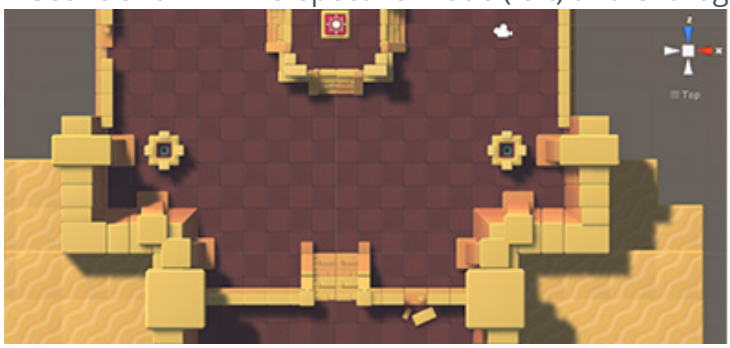
The Scene Gizmo has a conical arm on each side of the cube. The arms at the forefront are labelled **X**, **Y** and **Z**. Click on any of the conical axis arms to snap the Scene View Camera to the axis it represents (for example: top view, left view and front view). You can also right-click the cube to bring up a menu with a list of viewing angles. To return to the default viewing angle, right-click the Scene Gizmo and click **Free**.

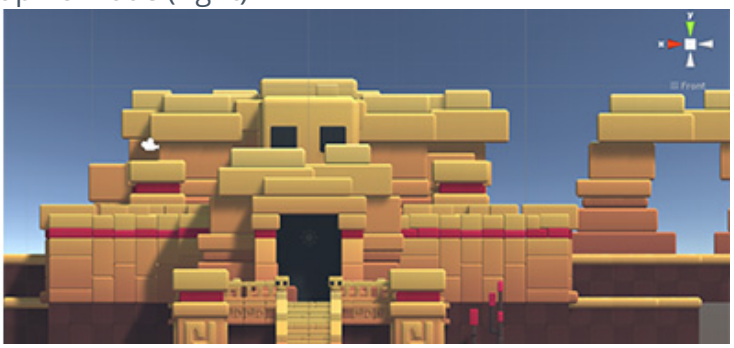
You can also toggle **Perspective** on and off. This changes the projection mode of the Scene View between **Perspective** and **Orthographic** (sometimes called 'isometric'). To do this, click the cube in the centre of the Scene Gizmo, or the text below it. Orthographic view has no perspective, and is useful in combination with clicking one of the conical axis arms to get a front, top or side elevation.





A Scene shown in Perspective mode (left) and Orthographic mode (right)





The same Scene viewed in top and front view, in orthographic mode

(Scene above from BITGEM)

If your Scene View is in an awkward viewpoint (upside-down, or just an angle you find confusing), Shift-click the cube at the centre of the Scene Gizmo to get back to a **Perspective** view with an angle that is looking at the Scene from the side and slightly from above.

Click on the padlock on the top right of the Scene Gizmo to enable or disable rotation of the Scene. Once Scene rotation is disabled, right-clicking the mouse pans the view instead of rotating it. This is the same as the Hand tool (see *Hand tool* , below).

Note that in **2D Mode** the Scene Gizmo is not displayed, because the only option is to have the view looking perpendicularly at the XY plane.

Scene Gizmo: Mac trackpad gestures

On a Mac with a trackpad, you can drag with two fingers to zoom the view.

You can also use three fingers to simulate the effect of clicking the arms of the **Scene Gizmo**: drag up, left, right or down to snap the Scene View Camera to the corresponding direction. In OS X 10.7 “Lion” you may have to change your trackpad settings in order to enable this feature:

Open System Preferences and then Trackpad (or type trackpad into Spotlight).
Click into the “More Gestures” option.
Click the first option labelled “Swipe between pages” and then either set it to “Swipe left or right with three fingers” or “Swipe with two or three fingers”.

Moving, orbiting and zooming in the Scene View

Moving, orbiting and zooming are key operations in Scene View navigation. Unity provides several ways to perform them for maximum accessibility:

Arrow movement

You can use the **Arrow Keys** to move around the Scene as though “walking” through it. The up and down arrows move the Camera forward and backward in the direction it is facing. The left and right arrows pan the view sideways. Hold down the **Shift** key with an arrow to move faster.

The Hand tool

When the Hand tool is selected (shortcut: **Q**), the following mouse controls are available:

Move: Click-drag to drag the Camera around.

Orbit: Hold **Alt**, and left-click and drag to orbit the Camera around the current pivot point. This option is not available in 2D mode, because the view is orthographic.

Zoom: Hold **Alt**, and right-click and drag to zoom the Scene View. On Mac you can also hold **Control**, and left-click and drag instead.

Hold down **Shift** to increase the rate of movement and zooming.

Flythrough mode

Use **Flythrough mode** to navigate the Scene View by flying around in first-person, similar to how you would navigate in many games.

Click and hold the right mouse button.

Move the view around using the mouse, the **WASD** keys to move left/right/forward/backward, and the **Q** and **E** keys to move up and down.
Hold down **Shift** to move faster.
Flythrough mode is designed for **Perspective Mode**. In **Orthographic Mode**, holding down the right mouse button and moving the mouse orbits the Camera instead.

Note that Flythrough mode is not available in 2D mode. Instead, holding the right mouse button down while moving the mouse pans around the Scene View.

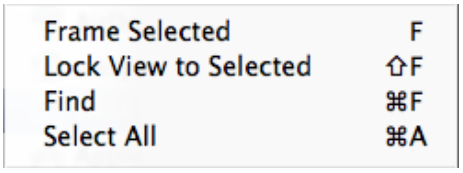
Movement shortcuts

For extra efficiency, all of these controls can also be used regardless of which transform tool is selected. The most convenient controls depend on which mouse or track-pad you are using:

Action	3-button mouse	2-button mouse or track-pad	Mac with only one mouse button or track-pad
Move	Hold Alt+middle mouse button, then drag	Hold Alt+Control+left-click, then drag	Hold Alt+Command+left-click, then drag
Orbit (Not available in 2D mode)	Hold Alt+left-click, then drag	Hold Alt+left-click, then drag	Hold Alt+left-click, then drag
Zoom	Use the scroll wheel, or hold Alt+right-click, then drag	Hold Alt+right-click, then drag	Use the two-finger swipe method to scroll in and out, or hold Alt-Control+left-click, then drag

Centering the view on a GameObject

To center the Scene View on a GameObject, select the GameObject in the Hierarchy, then move the mouse over the Scene View and press **F**. This feature can also be found in the menu bar under **Edit** > **Frame Selected**. To lock the view to the GameObject even when the GameObject is moving, press **Shift+F**. This feature can also be found in the menu bar under **Edit** > **Lock View to Selected**.



Positioning GameObjects

Leave feedback

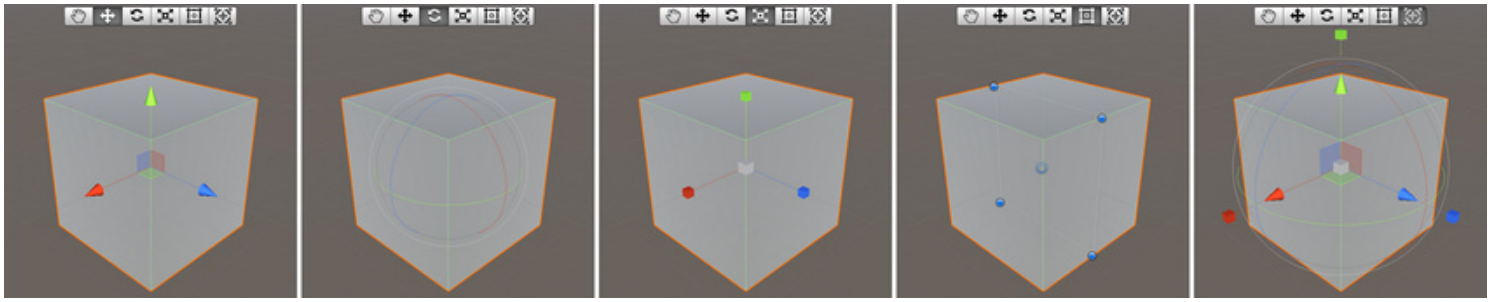
To select a **GameObject**, click on it in the Scene view or click its name in the Hierarchy window. To select or de-select multiple GameObjects, hold the **Shift** key while clicking, or drag a rectangle around multiple GameObjects to select them.

Selected GameObjects are highlighted in the Scene view. By default, this highlight is an orange outline around the GameObject; to change the highlight color and style, go to **Unity** > **Preferences** > **Color** and edit the **Selected Wireframe** and **Selected Outline** colours. See documentation on the Gizmo Menu for more information about the outline and wireframe selection visualizations. The selected GameObjects also display a **Gizmo** in the Scene view if you have one of the four Transform tools selected:

Move, Rotate, Scale, and RectTransform

The first tool in the **toolbar**, the **Hand Tool**, is for panning around the **Scene**. The **Move**, **Rotate**, **Scale**, **Rect Transform** and **Transform** tools allow you to edit individual GameObjects. To alter the Transform **component** of the GameObject, use the mouse to manipulate any Gizmo axis, or type values directly into the number fields of the **Transform component** in the **Inspector**.

Alternatively, you can select each of the four **Transform** modes with a hotkey: **W** for Move, **E** for Rotate, **R** for Scale, **T** for RectTransform, and **Y** for Transform.



The Move, Scale, Rotate, Rect Transform and Transform Gizmos

Move

At the center of the **Move** Gizmo, there are three small squares you can use to drag the GameObject within a single plane (meaning you can move two axes at once while the third keeps still).

If you hold shift while clicking and dragging in the center of the Move Gizmo, the center of the Gizmo changes to a flat square. The flat square indicates that you can move the GameObject around on a plane relative to the direction the Scene view **Camera** is facing.

Rotate

With the **Rotate** tool selected, change the GameObject's rotation by clicking and dragging the axes of the wireframe sphere Gizmo that appears around it. As with the Move Gizmo, the last axis you changed will be colored yellow. Think of the red, green and blue circles as performing rotation around the red, green and blue axes that appear in the Move mode (red is the x-axis, green in the y-axis, and blue is the z-axis). Finally, use the outermost circle to rotate the GameObject around the Scene view z-axis. Think of this as rotating in screen space.

Scale

The **Scale** tool lets you rescale the GameObject evenly on all axes at once by clicking and dragging on the cube at the center of the Gizmo. You can also scale the axes individually, but you should take care if you do this when there are child GameObjects, because the effect can look quite strange.

RectTransform

The RectTransform is commonly used for positioning 2D elements such as **Sprites** or UI elements, but it can also be useful for manipulating 3D GameObjects. It combines moving, scaling and rotation into a single Gizmo:

Click and drag within the rectangular Gizmo to move the GameObject.
Click and drag any corner or edge of the rectangular Gizmo to scale the GameObject.
Drag an edge to scale the GameObject along one axis.
Drag a corner to scale the GameObject on two axes.
To rotate the GameObject, position your cursor just beyond a corner of the rectangle. The cursor changes to display a rotation icon. Click and drag from this area to rotate the GameObject.
Note that in 2D mode, you can't change the z-axis in the Scene using the Gizmos. However, it is useful for certain scripting techniques to use the z-axis for other purposes, so you can still set the z-axis using the Transform component in the Inspector.

For more information on transforming GameObjects, see documentation on the Transform Component.

Transform

The **Transform** tool combines the **Move**, **Rotate** and **Scale** tools. Its Gizmo provides handles for movement and rotation. When the **Tool Handle Rotation** is set to **Local** (see below), the Transform tool also provides handles for scaling the selected GameObject.

Gizmo handle position toggles

The **Gizmo handle position toggles** are used to define the location of any Transform tool Gizmo, and the handles use to manipulate the Gizmo itself.

Gizmo display toggles

For position

Click the **Pivot**/**Center** button on the left to toggle between **Pivot** and **Center**.

Pivot positions the Gizmo at the actual pivot point of a **Mesh**.
Center positions the Gizmo at the center of the GameObject's rendered bounds.

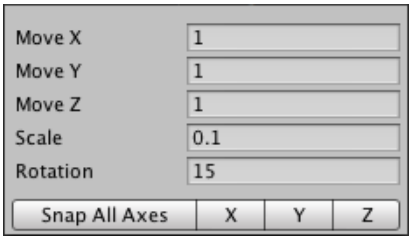
For rotation

Click the **Local**/**Global** button on the right to toggle between **Local** and **Global**.

Local keeps the Gizmo's rotation relative to the GameObject's.
Global clamps the Gizmo to world space orientation.

Unit snapping

While dragging any Gizmo Axis using the **Move** tool or the **Transform** tool, hold the **Control** key (**Command** on Mac) to snap to increments defined in the **Snap Settings** (menu: **Edit** > **Snap Settings...**)



Surface snapping

While dragging in the center using the **Move** tool or the **Transform** tool, hold **Shift** and **Control** (**Command** on Mac) to quickly snap the GameObject to the intersection of any **Collider**.

Look-at rotation

While using the **Rotate** tool or the **Transform** tool, hold **Shift** and **Control** (**Command** on Mac) to rotate the GameObject towards a point on the surface of any **Collider**.

Vertex snapping

Use **vertex snapping** to quickly assemble your Scenes: take any vertex from a given Mesh and place that vertex in the same position as any vertex from any other Mesh you choose. For example, use vertex snapping to align road sections precisely in a racing game, or to position power-up items at the vertices of a Mesh.

Follow the steps below to use vertex snapping:

Select the Mesh you want to manipulate and make sure the **Move** tool or the **Transform** tool is active.

Press and hold the **V** key to activate the vertex snapping mode.

Move your cursor over the vertex on your Mesh that you want to use as the pivot point.

Hold down the left mouse button once your cursor is over the vertex you want and drag your Mesh next to any other vertex on another Mesh.

Release the mouse button and the **V** key when you are happy with the results (**Shift+V** acts as a toggle of this functionality).

Note: You can snap vertex to vertex, vertex to surface, and pivot to vertex.

Screen Space Transform

While using the **Transform** tool, hold down the **Shift** key to enable Screen Space mode. This mode allows you to move, rotate and scale GameObjects as they appear on the screen, rather than in the Scene.

Transform tool added in 2017.3

2017–05–18 Page amended with editorial review

Scene view Control Bar

Leave feedback

The **Scene view** control bar lets you choose various options for viewing the **Scene** and also control whether lighting and audio are enabled. These controls only affect the **Scene view** during development and have no effect on the built game.

Draw mode menu

The first drop-down menu selects which **Draw Mode** will be used to depict the Scene. The available options are:

Shading mode

Shaded: show surfaces with their textures visible.
Wireframe: draw meshes with a wireframe representation.
Shaded Wireframe: show meshes textured and with wireframes overlaid.

Miscellaneous

Shadow Cascades: show directional light shadow cascades.
Render Paths: show the rendering path for each object using a color code: Blue indicates deferred shading, Green indicates deferred lighting, yellow indicates forward rendering and red indicates vertex lit.
Alpha Channel: render colors with alpha.
Overdraw: render objects as transparent "silhouettes". The transparent colors accumulate, making it easy to spot places where one object is drawn over another.
Mipmaps: show ideal texture sizes using a color code: red indicates that the texture is larger than necessary (at the current distance and resolution); blue indicates that the texture could be larger. Naturally, ideal texture sizes depend on the resolution at which the game will run and how close the **camera** can get to particular surfaces.

Deferred

These modes let you view each of the elements of the G-buffer (**Albedo**, **Specular**, **Smoothness** and **Normal**) in isolation. See documentation on Deferred Shading for more information.

Global Illumination

The following modes are available to help visualise aspects of the Global Illumination system: **UV Charts**, **Systems**, **Albedo**, **Emissive**, **Irradiance**, **Directionality**, **Baked**, **Clustering** and **Lit Clustering**. See documentation on GI Visualisations for informaiton about each of these modes.

Material Validator

There are two **Material Validator** modes: **Albedo** and **Metal Specular**. These allow you to check whether your physically-based materials use values within the recommended ranges. See Physically Based Material Validator for more information.

2D, lighting and Audio switches

To the right of the **Render Mode** menu are three buttons that switch certain Scene view options on or off:

2D: switches between 2D and 3D view for the Scene. In 2D mode the camera is oriented looking towards positive z, with the x axis pointing right and the y axis pointing up.
Lighting: turns Scene view lighting (lights, object shading, etc) on or off.
Audio: turns Scene view **audio effects** on or off.

Effects button and menu

The menu (activated by the small mountain icon to the right of the **Audio** button) has options to enable or disable **rendering** effects in the Scene view.

Skybox: a skybox texture rendered in the Scene's background
Fog: gradual fading of the view to a flat color with distance from the camera.

Flares: **lens flares** on lights.
Animated Materials: Defines whether or not animated materials show the animation
The **Effects** button itself acts as a switch that enables or disables all the effects at once.

Gizmos menu

The **Gizmos** menu contains lots of options for how objects, icons, and **gizmos** are displayed. This menu is available in both the Scene view and the Game view. See documentation on the Gizmos Menu manual page for more information.

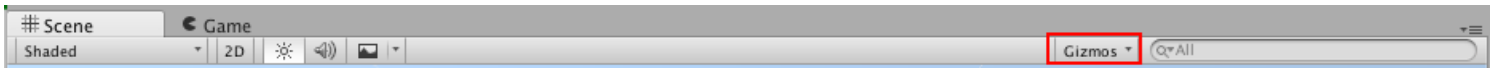
Search box

The rightmost item on the control bar is a search box that lets you filter items in the Scene view by their names and/or types (you can select which with the small menu at the left of the search box). The set of items that match the search filter are also be shown in the Hierarchy view which, by default, is located to the left of the Scene view.

Gizmos menu

Leave feedback

The Scene view and the Game view both have a **Gizmos** menu. Click the **Gizmos** button in the **toolbar** of the **Scene** view or the Game view to access the **Gizmos** menu.



The **Gizmos** button in the Scene view

The Gizmos menu at the top of the Scene view and Game view window

Property	Function
3D Icons	The **3D Icons** checkbox controls whether component icons (such as those for Lights and Cameras) are drawn by the Editor in 3D in the Scene view. When the **3D Icons** checkbox is ticked, component icons are scaled by the Editor according to their distance from the Camera, and obscured by GameObjects in the Scene. Use the slider to control their apparent overall size. When the **3D Icons** checkbox is not ticked, component icons are drawn at a fixed size, and are always drawn on top of any GameObjects in the Scene view. See Gizmos and Icons, below, for images and further information.
Show Grid	The **Show Grid** checkbox switches the standard Scene measurement grid on (checked) and off (unchecked) in the Scene view. To change the colour of the grid, go to **Unity** > **Preferences** > **Colors** and alter the **Grid** setting. This option is only available in the Scene view Gizmos menu; you cannot enable it in the Game view Gizmos menu. See Show Grid, below, for images and further information.

Property	Function
Selection Outline	Check **Selection Outline** to show the selected GameObjects with a colored outline around them. If the selected GameObject extends beyond the edges of the Scene view, the outline is cropped to follow the edge of the window. To change the colour of the Selection Outline, go to **Unity** > **Preferences** > **Colors** and alter the **Selected Outline** setting. This option is only available in the Scene view Gizmos menu; you cannot enable it in the Game view Gizmos menu. See Selection Outline and Selection Wire, below, for images and further information.
Selection Wire	Check **Selection Wire** to show the selected GameObjects with their wireframe Mesh visible. To change the colour of the Selection Wire, go to **Unity** > **Preferences** > **Colors** and alter the **Selected Wireframe** setting. This option is only available in the Scene view Gizmos menu; you cannot enable it in the Game view Gizmos menu. See Selection Outline and Selection Wire, below, for images and further information.
Built-in Components	The **Built-in Components** list controls the visibility of the icons and Gizmos of all component types that have an icon or Gizmo. See Built-in Components, below, for further information.

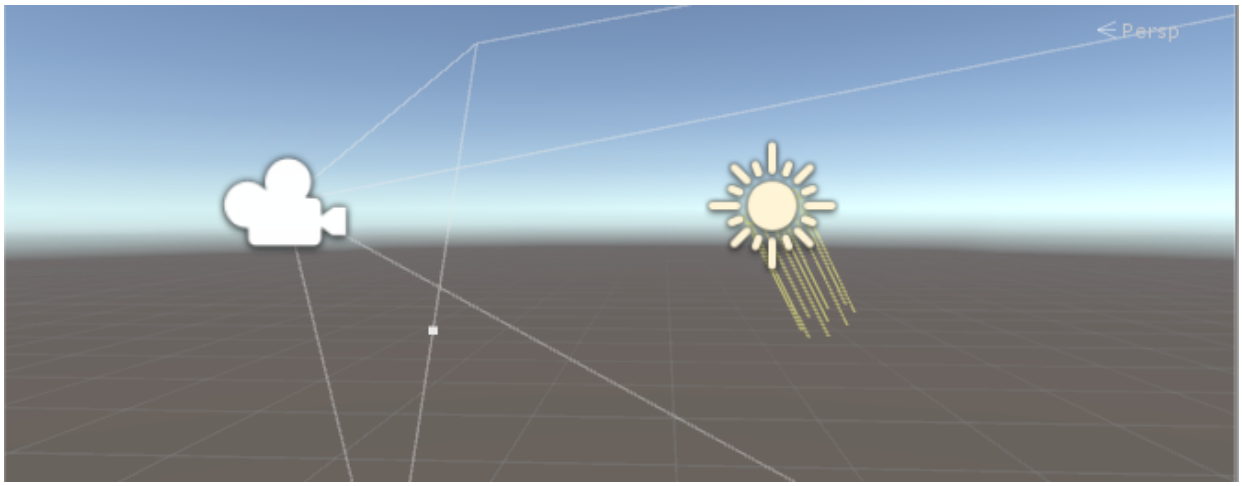
Gizmos and Icons

Gizmos

Gizmos are graphics associated with **GameObjects** in the Scene. Some Gizmos are only drawn when the GameObject is selected, while other Gizmos are drawn by the Editor regardless of which **GameObjects** are selected. They are usually wireframes, drawn with code rather than bitmap graphics, and can be interactive. The **Camera** Gizmo and **Light direction** Gizmo (shown below) are both examples of built-in Gizmos; you can also create your own Gizmos using script. See documentation on Understanding Frustum for more information about the Camera.

Some Gizmos are passive graphical overlays, shown for reference (such as the **Light direction** Gizmo, which shows the direction of the light). Other Gizmos are interactive, such as the AudioSource **spherical range** Gizmo, which you can click and drag to adjust the maximum range of the AudioSource.

The **Move**, **Scale**, **Rotate** and **Transform** tools are also interactive Gizmos. See documentation on Positioning GameObjects to learn more about these tools.

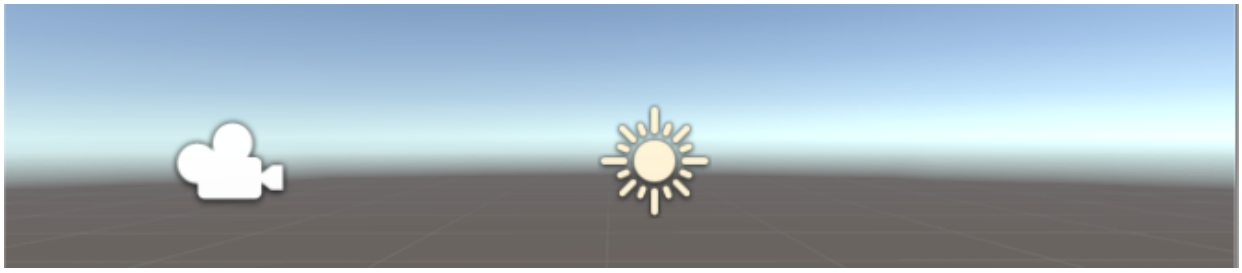


The Camera Gizmo and a Light Gizmo. These Gizmos are only visible when they are selected.

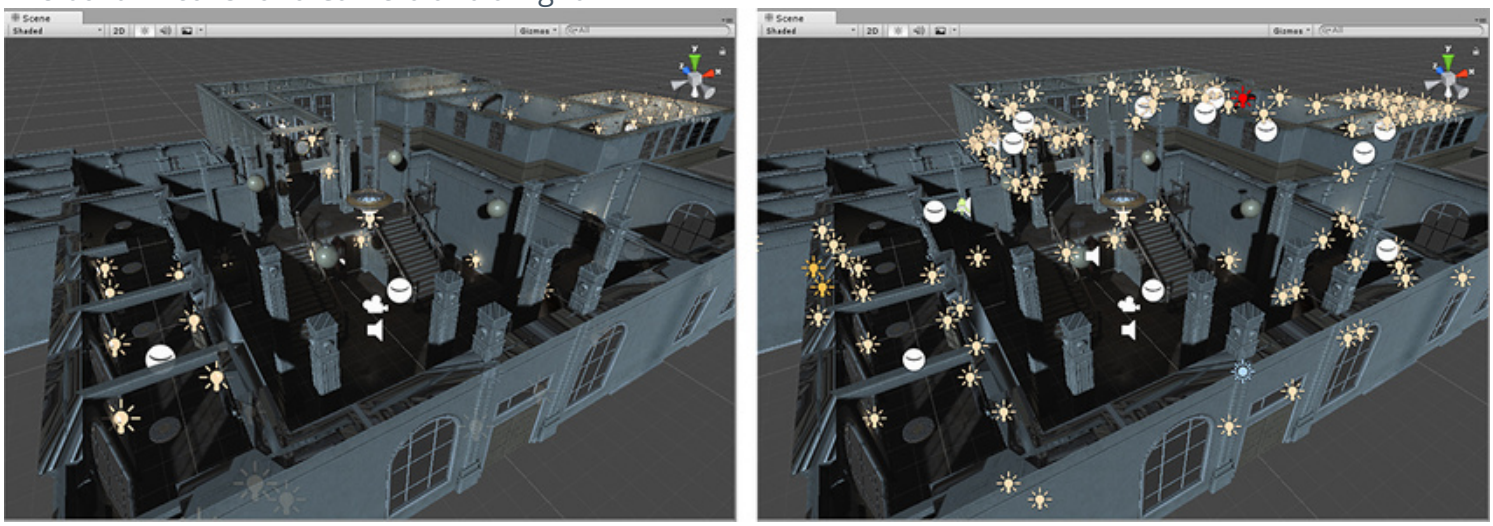
See the Script Reference page for the OnDrawGizmos function for further information about implementing custom Gizmos in your **scripts**.

Icons

You can display **icons** in the Game view or Scene view. They are flat, billboard-style overlays which you can use to clearly indicate a GameObject's position while you work on your game. The **Camera** icon and **Light** icon are examples of built-in icons; you can also assign your own to GameObjects or individual scripts (see documentation on Assigning Icons to lean how to do this).



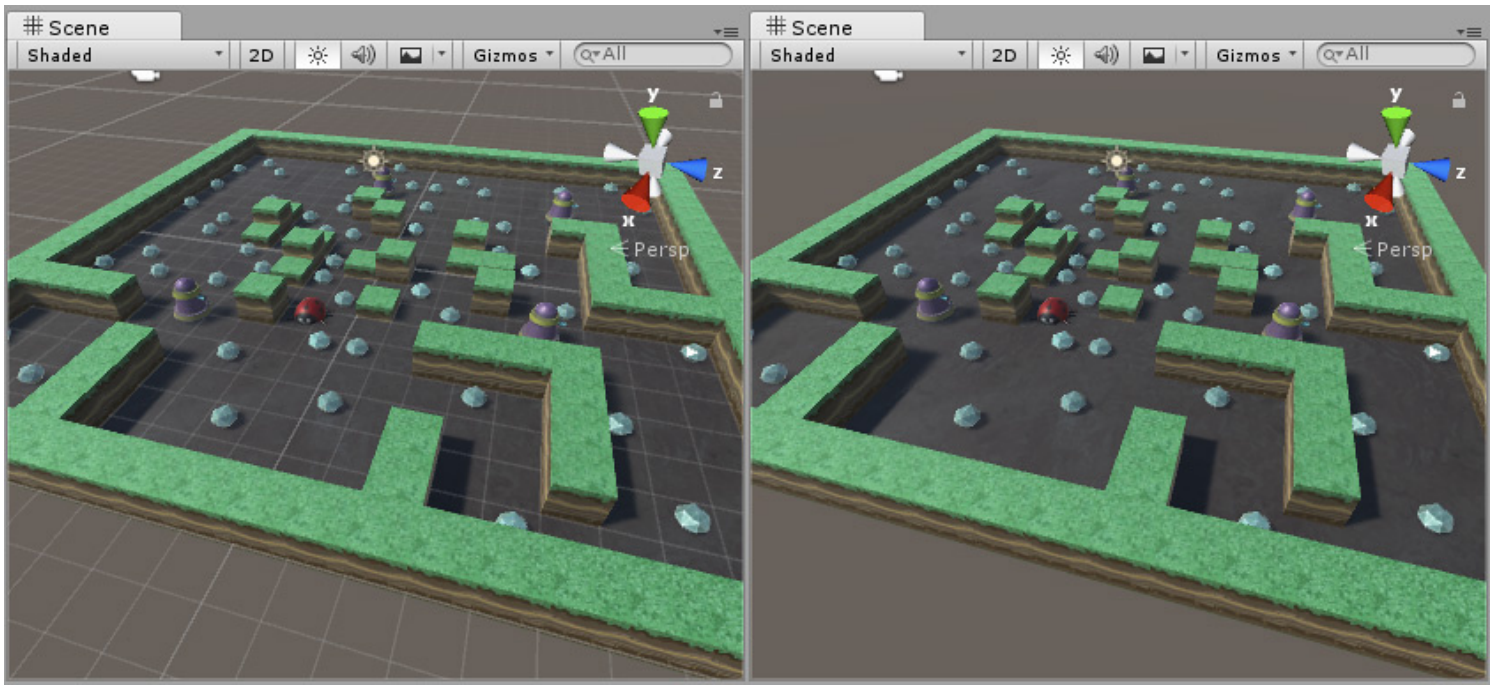
The built-in icons for a Camera and a Light



Left: icons in 3D mode. **Right**: icons in 2D mode.

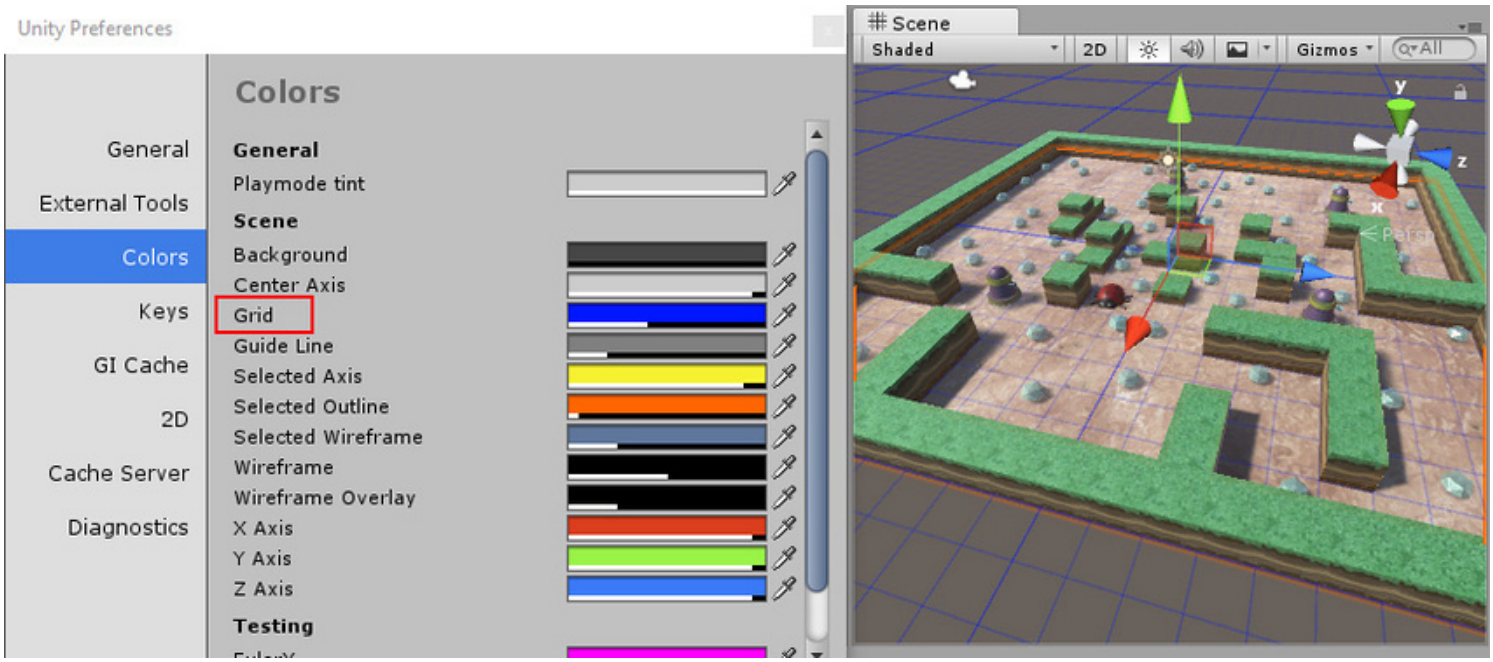
Show Grid

The **Show Grid** feature toggles a grid on the plane of your Scene. The following images show how this appears in the Scene view:



Left: Scene view grid is enabled. **Right**: Scene view grid is disabled.

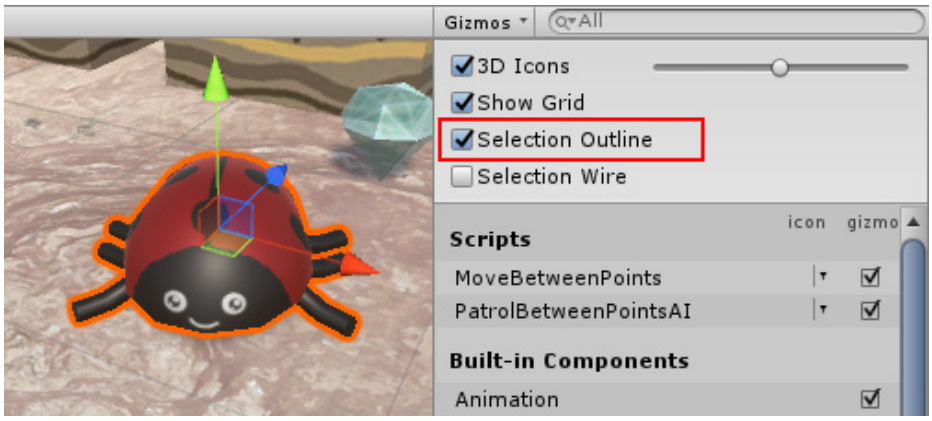
To change the colour of the grid, go to **Unity** > **Preferences** > **Colors** and alter the **Grid** setting. In this image, the Scene view grid is colored dark blue so that it shows up better against the light-colored floor:



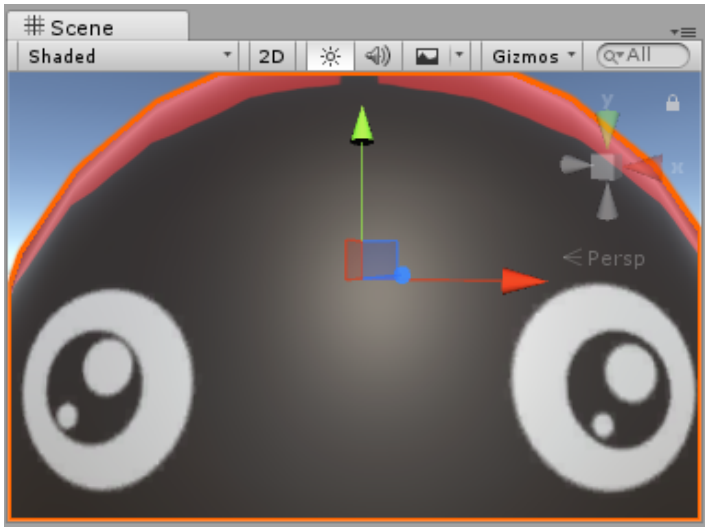
Selection Outline and Selection Wire

Selection Outline

When **Selection Outline** is enabled, then when you select a GameObject in the Scene view or Hierarchy window, an orange outline appears around that GameObject in the Scene view:

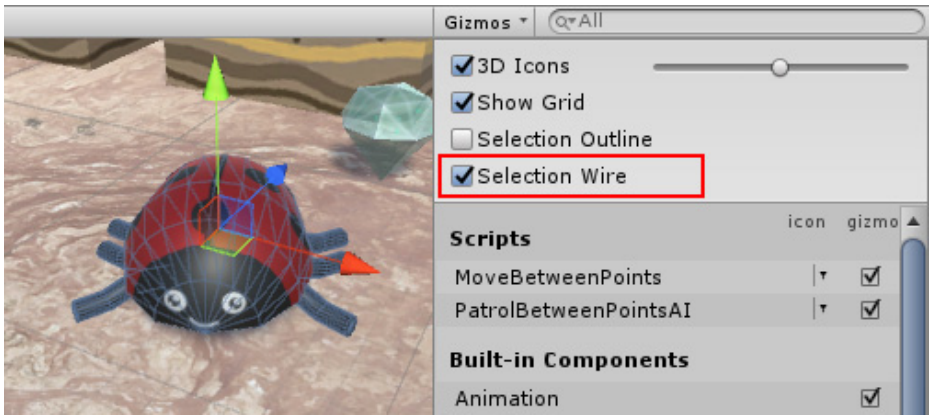


If the selected GameObject fills most of the Scene view and extends beyond the edges of the window, the selection outline runs along the edge of the window:



Selection Wire

When **Selection Wire** is enabled, then when you select a GameObject in the Scene view or Hierarchy window, the wireframe **Mesh** for that GameObject is made visible in the Scene view:



Selection colors

You can set custom colours to selection wireframes; to do this, go to **Unity** > **Preferences** > **Colors** and alter the **Selected Outline** setting to change the **Selection Outline**, or the **Selected Wireframe** to change the **Selection Wire** setting.

Built-in Components

Use the **Built-in Components** list to control the visibility of the icons and Gizmos of all **component** types that have an icon or Gizmo.

Some built-in component types (such as Rigidbody) are not listed here because they do not have an icon or Gizmo shown in the Scene view. Only components which have an icon or a Gizmo are listed.

The Editor also lists some of your project scripts here, above the built-in components. These are:

Scripts with an icon assigned to them (see documentation on Assigning icons).

Scripts which implement the OnDrawGizmos function.

Scripts which implement the OnDrawGizmosSelected function.

Recently changed items are at the top of the list.

The Gizmos menu, displaying some items with custom icons assigned and some recently modified items

The **icon** column shows or hides the icons for each listed type of component. Click the small icon under the **icon** column to toggle visibility of that icon. If the icon is in full colour in the menu then it is displayed in the Scene view; if it is greyed out in the menu, then it is not visible in the Scene view. Any scripts with custom icons show a small drop-down menu arrow. Click this to show the icon selector menu, where you can change the script's icon.

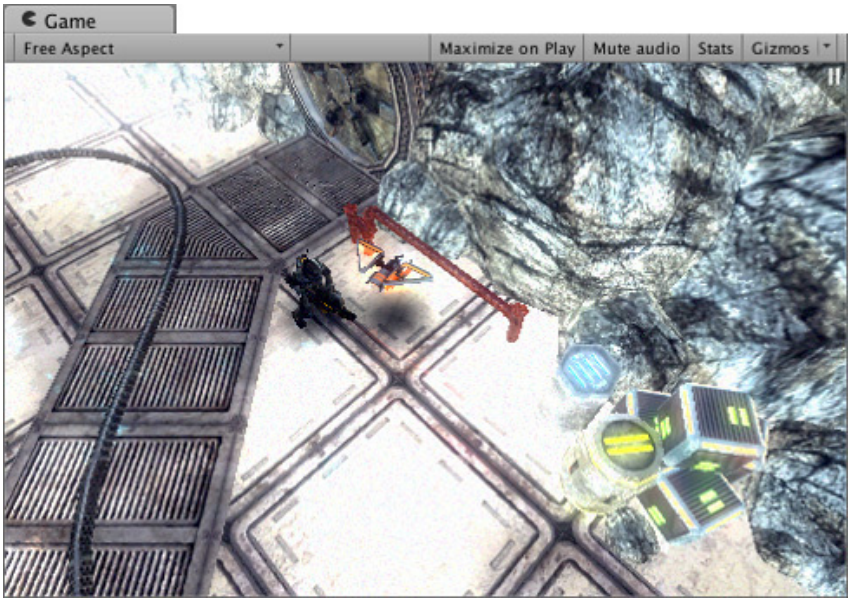
Note: If an item in the list has a Gizmo but no icon, there is no option in the icon column.

Tick the checkboxes in the **Gizmo** column to select whether Gizmo graphics are drawn by the Editor for a particular component type. For example, **Colliders** have a predefined wireframe Gizmo to show their shape, and Cameras have a Gizmo which shows the view frustum. Your own scripts can draw custom Gizmos appropriate to their purpose; implement OnDrawGizmos or OnDrawGizmosSelected to do this. Uncheck the checkboxes in this column to turn these Gizmos off.

Note: If an item in the list has an icon but no Gizmo, there is no checkbox in this column.

The Game view

Leave feedback

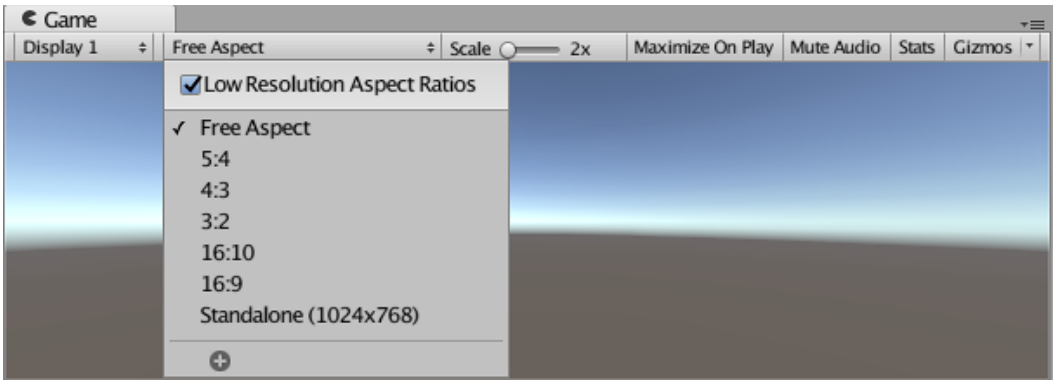


The Game View is rendered from the Camera(s) in your game. It is representative of your final, published game. You will need to use one or more **Cameras** to control what the player actually sees when they are playing your game. For more information about **Cameras**, please view the Camera Component page.

Play mode

Use the buttons in the **Toolbar** to control the Editor Play Mode and see how your published game plays. While in Play Mode, any changes you make are temporary, and will be reset when you exit Play Mode. The Editor **UI** darkens to remind you of this.

Game view Control Bar



Button: Function:

Display	Click this to choose from a list of Cameras if you have multiple Cameras in the **Scene**. This is set to **Display 1** by default. (You can assign Displays to cameras in the Camera module, under the **Target Display** drop-down menu.)
Aspect	Select different values to test how your game will look on monitors with different **aspect ratios**. This is set to **Free Aspect** by default.
Low Resolution Aspect Ratios	Check this box if you want to emulate the **pixel** density of older displays: This reduces the resolution of the Game view when an aspect ratio is selected. It is always enabled when the Game view is on a non-Retina display.

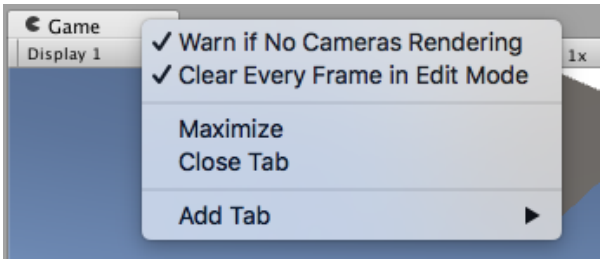
Button:	*Function:*
Scale slider	Scroll right to zoom in and examine areas of the Game screen in more detail. It also allows you to zoom out to see the entire screen where the device resolution is higher than the Game view window size. You can also use the scroll wheel and middle mouse button to do this while the game is stopped or paused.
Maximize on Play	Click to enable: Use this to make the Game view maximize (100% of your Editor Window) for a full-screen preview when you enter Play Mode.
Mute audio	Click to enable: Use this to mute any in-game audio when you enter Play Mode.
Stats	Click this to toggle the Statistics overlay, which contains Rendering Statistics about your game's audio and graphics. This is very useful for monitoring the performance of your game while in Play Mode.
Gizmos	Click this to toggle the visibility of Gizmos. To only see certain types of Gizmo during Play Mode, click the drop-down arrow next to the word **Gizmos** and only check the boxes of the Gizmo types you want to see. (See Gizmos Options below.)

Gizmos menu

The Gizmos Menu contains lots of options for how objects, icons, and gizmos are displayed. This menu is available in both the **Scene view** and the Game view. See documentation on the Gizmos Menu for more information.

Advanced options

Right-click the Game tab to display advanced Game View options.



Warn if No Cameras Rendering: This option is enabled by default: It causes a warning to be displayed if no Cameras are **rendering** to the screen. This is useful for diagnosing problems such as accidentally deleting or disabling a Camera. Leave this enabled unless you are intentionally not using Cameras to render your game.

Clear Every Frame in Edit Mode: This option is enabled by default: It causes the Game view to be cleared every frame when your game is not playing. This prevents smearing effects while you are configuring your game. Leave this enabled unless you are depending on the previous frame's contents when not in Play Mode.

2018–04–25 Page amended with limited editorial review

Low Resolution Aspect Ratios Game view option available on Windows from 2018.2

The Hierarchy window

Leave feedback

The default Hierarchy window view when you open a new Unity project

The **Hierarchy** window contains a list of every **GameObject** (referred to in this guide as an "object") in the current **Scene**. Some of these are direct instances of **Asset** files (like 3D models), and others are instances of **Prefabs**, which are custom objects that make up most of your game. As objects are added and removed in the Scene, they will appear and disappear from the Hierarchy as well.

By default, objects are listed in the Hierarchy window in the order they are made. You can re-order the objects by dragging them up or down, or by making them "child" or "parent" objects (see below).

Parenting

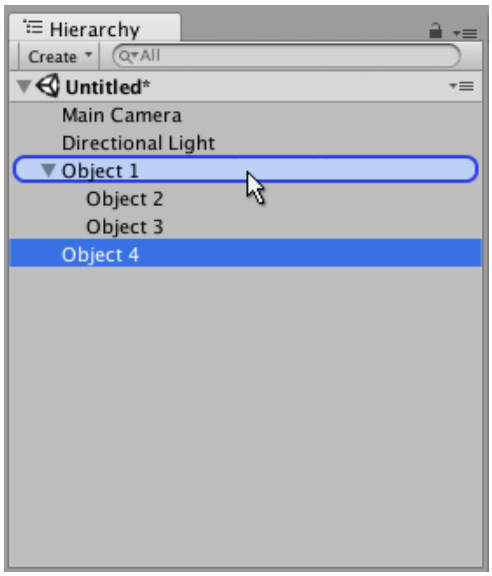
Unity uses a concept called **Parenting**. When you create a group of objects, the topmost object or Scene is called the "parent object", and all objects grouped underneath it are called "child objects" or "children". You can also created nested parent-child objects (called "descendants" of the top-level parent object).

In this image, **Child** and **Child 2** are the child objects of **Parent**. **Child 3** is a child object of **Child 2**, and a descendant object of **Parent**.

Click a parent object's drop-down arrow (on the left-hand side of its name) to show or hide its children. Hold down the Alt key while clicking the drop-down arrow to toggle visibility of all descendant objects of the parent, in addition to the immediate child object.

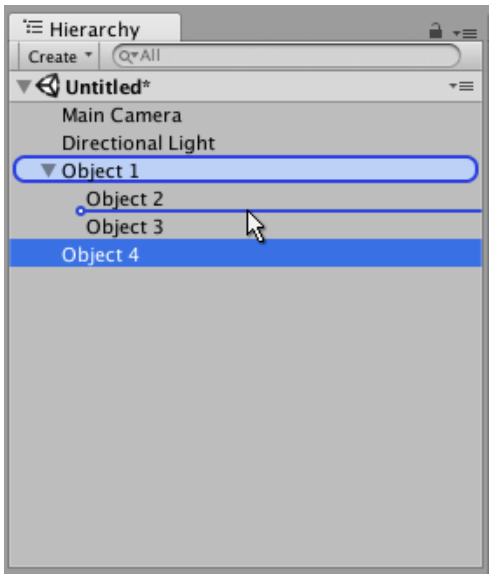
Making a child object

To make any object the “child” of another, drag and drop the intended child object onto the intended parent object in the Hierarchy.



In this image, **Object 4** (selected) is being dragged onto the intended parent object, **Object 1** (highlighted in a blue capsule).

You can also drag-and-drop an object alongside other objects to make them “siblings” - that is, child objects under the same parent object. Drag the object above or below an existing object until a horizontal blue line appears, and drop it there to place it alongside the existing object.



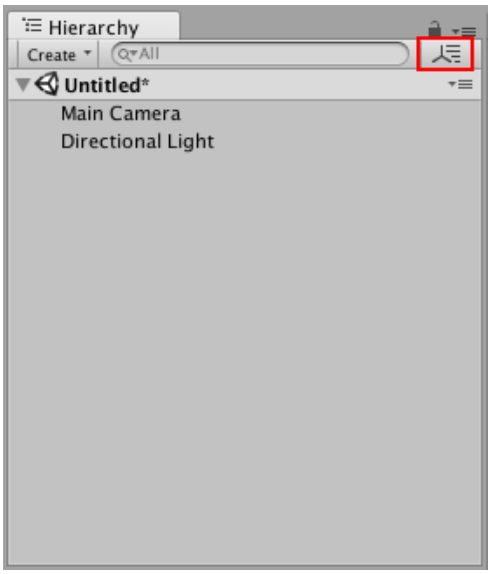
In this image, **Object 4** (selected) is being dragged between **Object 2** and **Object 3** (indicated by the blue horizontal line), to be placed here as a sibling of these two objects under the parent object **Object 1** (highlighted in a blue capsule).

Child objects inherit the movement and rotation of the parent object. To learn more about this, see documentation on the Transform component.

Alphanumeric sorting

The order of objects in the Hierarchy window can be changed to alphanumeric order. In the menu bar, select **Edit > Preferences** in Windows or **Unity > Preferences** in OS X to launch the **Preferences** window. Check **Enable Alpha Numeric Sorting**.

When you check this, an icon appears in the top-right of the Hierarchy window, allowing you to toggle between **Transform** sorting (the default value) or **Alphabetic** sorting.



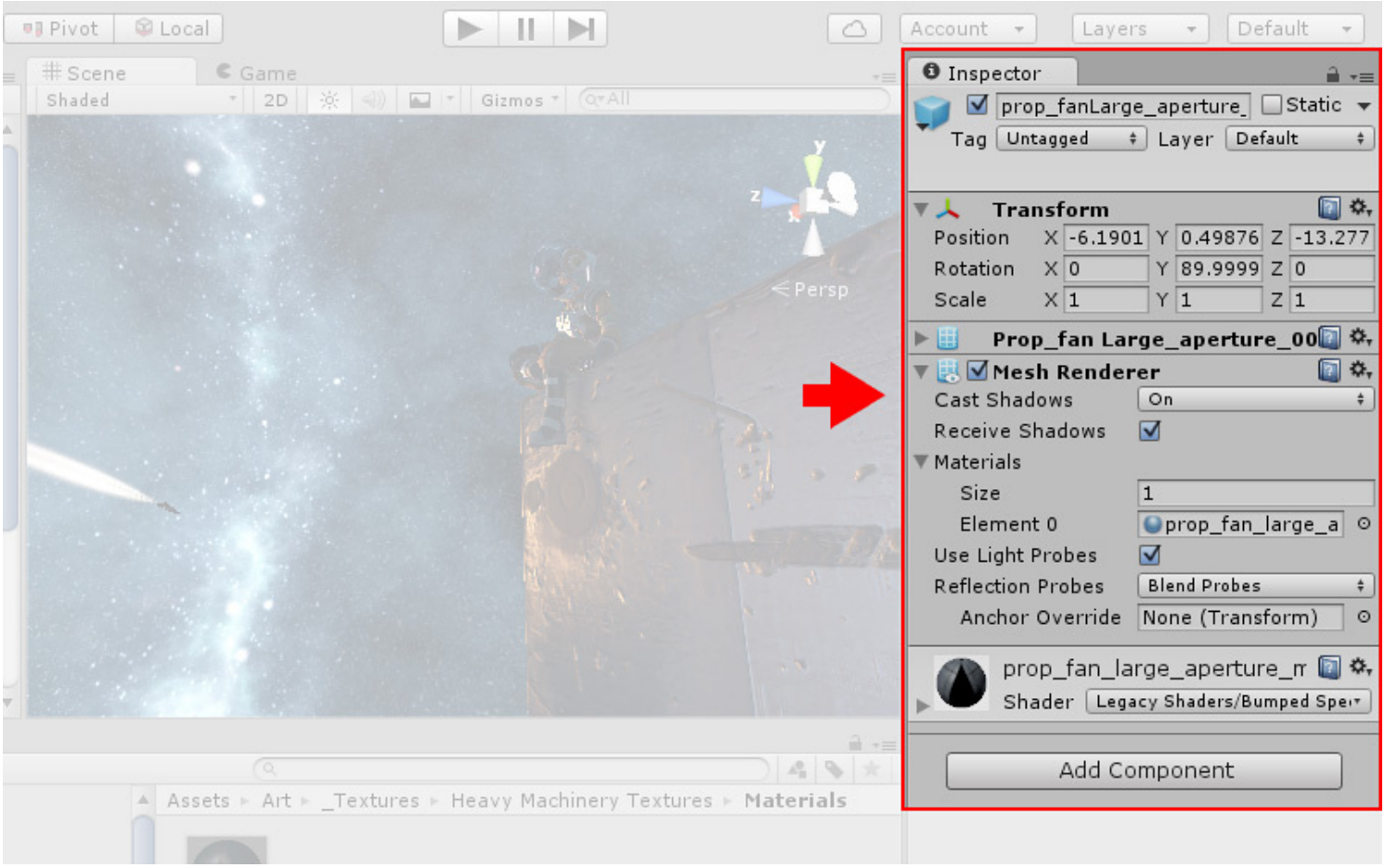
Multi-Scene editing

It is possible to have more than one Scene open in the Hierarchy window at the same time. To find out more about this, see the Multi Scene Editing page.

The Inspector window

Leave feedback

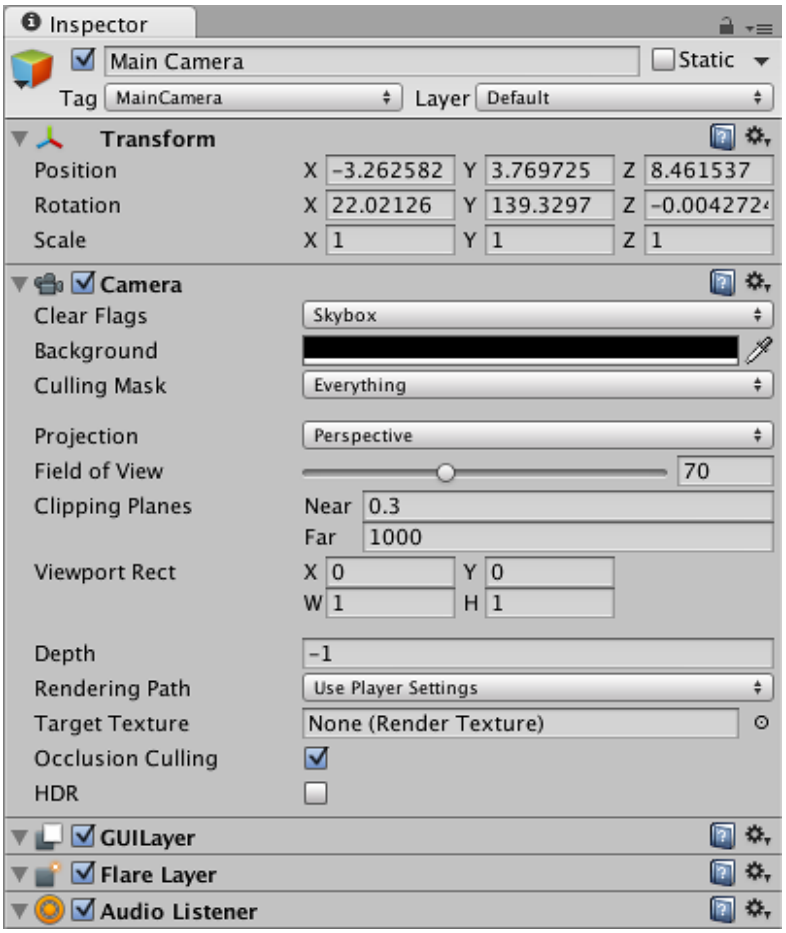
Projects in the Unity Editor are made up of multiple GameObjects that contain **scripts**, sounds, Meshes, and other graphical elements such as Lights. The **Inspector** window (sometimes referred to as "the Inspector") displays detailed information about the currently selected GameObject, including all attached components and their properties, and allows you to modify the functionality of GameObjects in your **Scene**.



The Inspector in its default position in Unity

Inspecting GameObjects

Use the Inspector to view and edit the properties and settings of almost everything in the Unity Editor, including physical game items such as GameObjects, Assets, and Materials, as well as in-Editor settings and preferences.

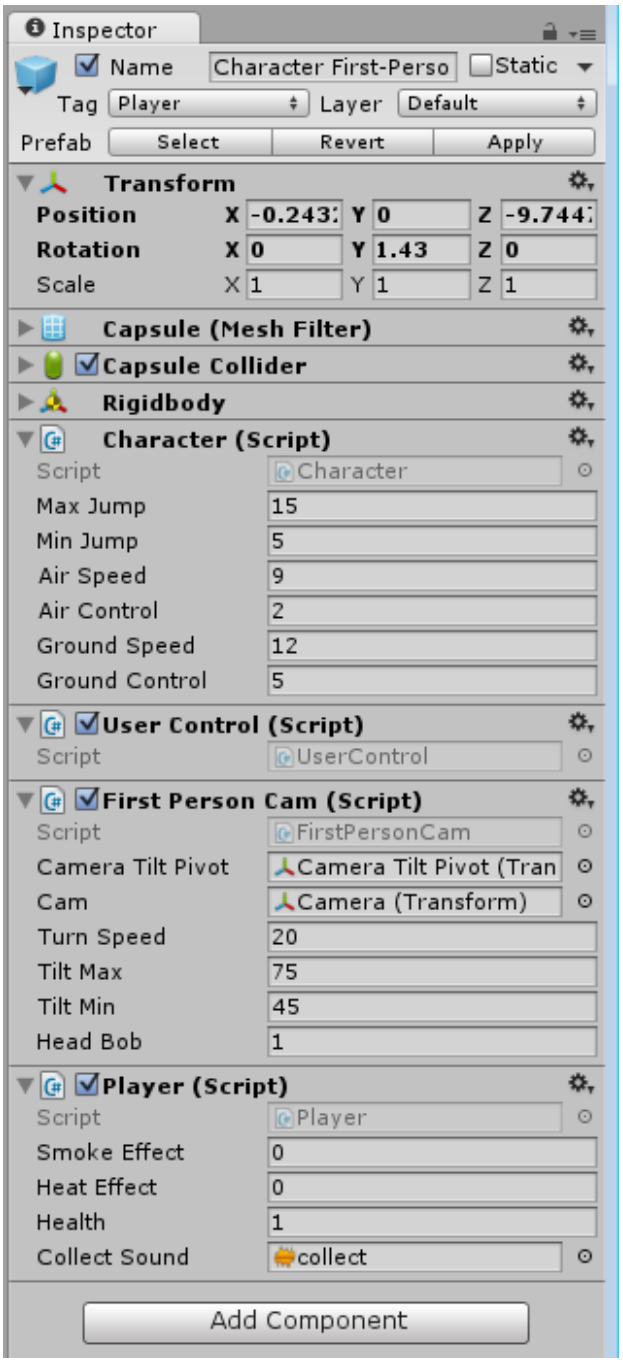


The Inspector window displaying settings for a typical GameObject and its components

When you select a GameObject in either the Hierarchy or Scene view, the Inspector shows the properties of all components and Materials of that GameObject. Use the Inspector to edit the settings of these components and Materials.

The image above shows the Inspector with the Main **Camera** GameObject selected. In addition to the GameObject's Position, Rotation, and Scale values, all the properties of the Main **Camera** are available to edit.

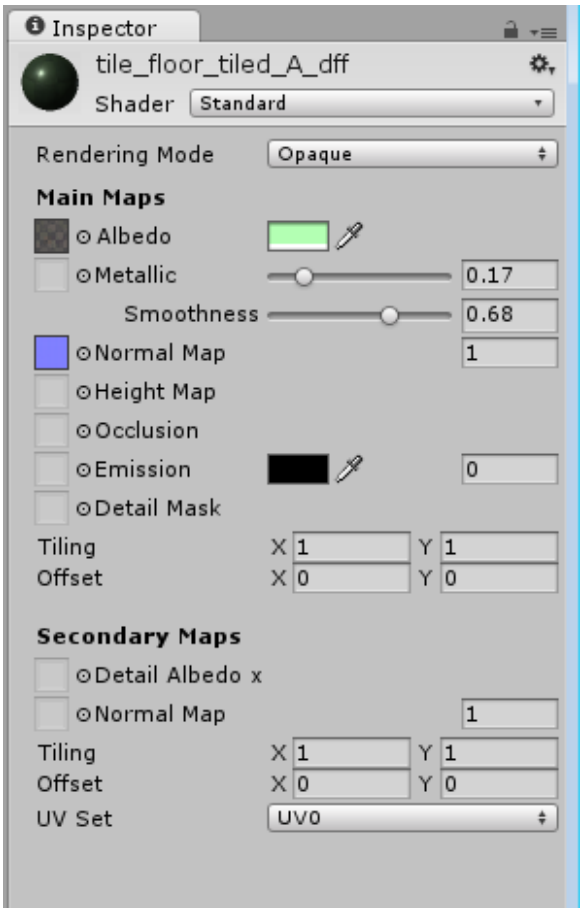
Inspecting script variables



The Inspector window displaying settings for a GameObject with several custom scripts attached. The scripts' public properties are available to edit

When GameObjects have custom script components attached, the Inspector displays the public variables of that script. You can edit these variables as settings in the same way you can edit the settings of the Editor's built-in components. This means that you can set parameters and default values in your scripts easily without modifying the code.

Inspecting Assets

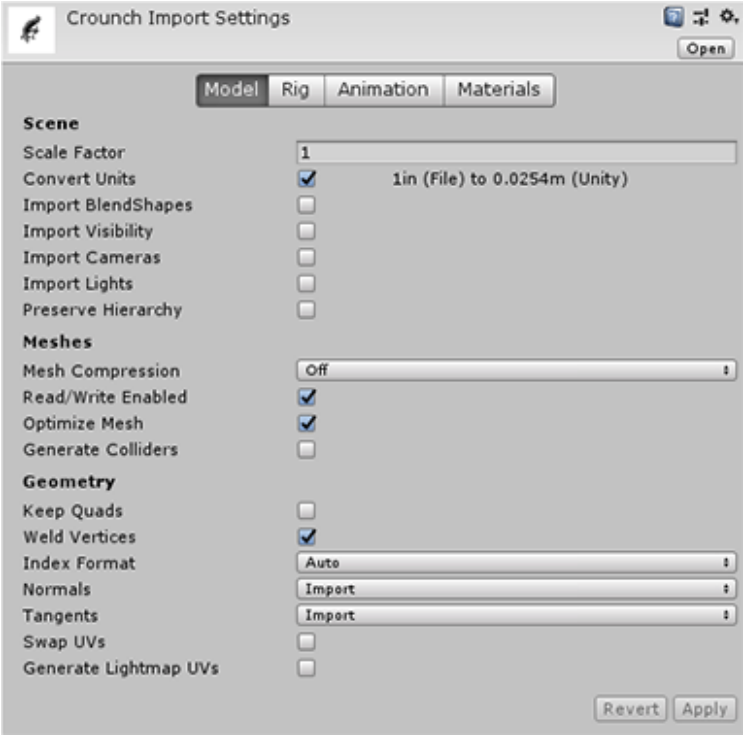


The Inspector window displaying the settings for a Material Asset

When an Asset is selected in your Project window, the Inspector shows you the settings related to how that Asset is imported and used at run time (when your game is running either in the Editor or your published build).

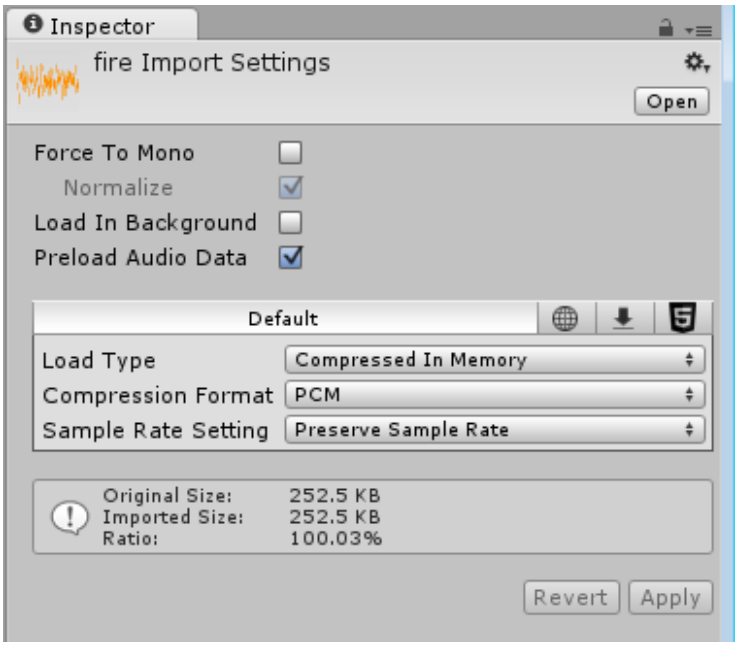
Each type of Asset has a different selection of settings. The images below demonstrate some examples of the Inspector displaying the import settings for other Asset types:

The Model tab of the Model Import Settings window:



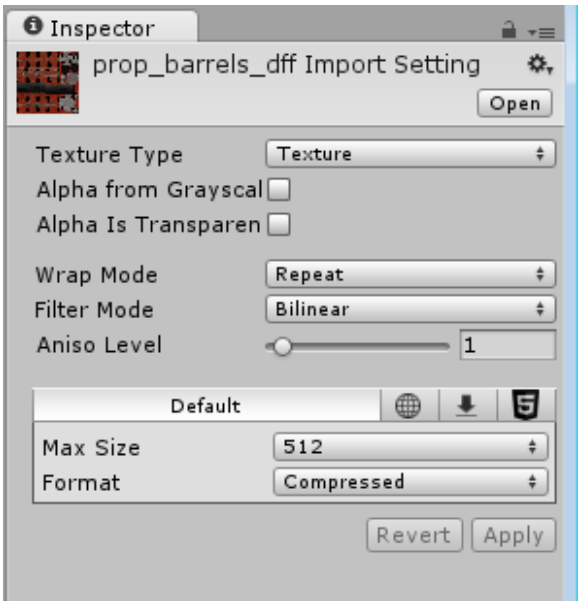
The Inspector window displaying the import settings for an .fbx file containing 3D Models

The Audio Clip Import Settings window:



The Inspector window displaying the import settings for an Audio file

The Texture Import Setting window:



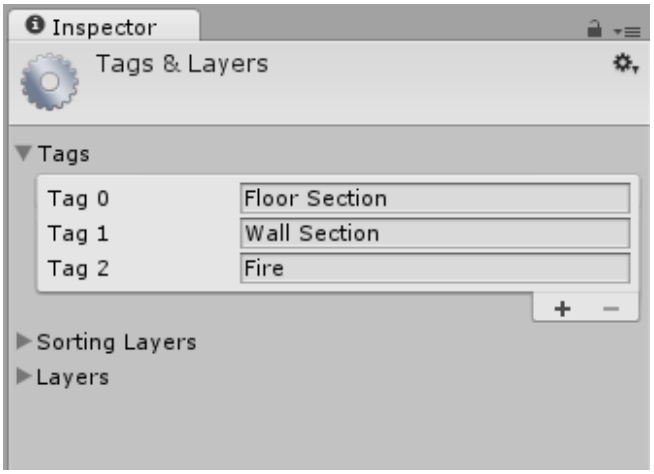
The Inspector window displaying the Import Settings for a Texture

Prefabs

If you have a Prefab selected, some additional options are available in the Inspector window.

For more information, see documentation on Prefabs.

Project settings



The Inspector window displaying the **Tags & Layers** Project Settings panel

When you select any of the Project Settings categories (menu: **Editor** > **Project Settings**), these settings are displayed in the Inspector window. For more information, see documentation on Project Settings.

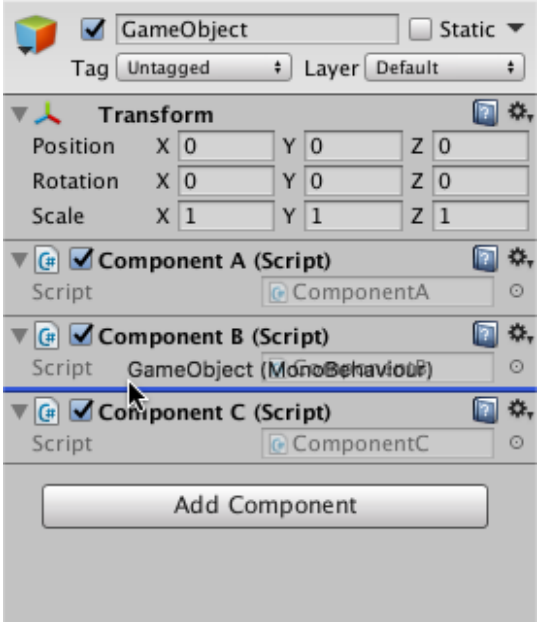
Icons and labels

You can assign custom icons to GameObjects and scripts. These display in the Scene view along with built-in icons for GameObjects such as Lights and Cameras.

For more about icons and labels, see Unity documentation on assigning icons.

Re-ordering components

To reorder components in the Inspector window, drag-and-drop their headers from one position to another. When you drag a component header, a blue insertion marker appears. This shows you where the component should go when you drop the header:



Dragging and dropping components on a GameObject in the Inspector window

You can only reorder components on a GameObject. You can't move components between different GameObjects.

You can also drag and drop script Assets directly into the position you want them to appear.

When you select multiple GameObjects, the Inspector displays all of the components that the selected GameObjects have in common. To reorder all of these common components at once, multi-select the GameObjects, then drag-and-drop the components into a new position in the Inspector.

The order you give to components in the Inspector window is the order you need to use when querying components in your user scripts. If you query the components programmatically, you'll get the order you see in the Inspector.

2018–04–23 Page amended with limited editorial review

Component drag and drop added in Unity 5.6

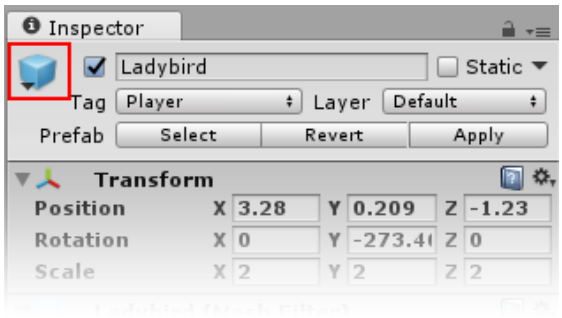
Assigning icons

Leave feedback

Unity allows you to assign custom icons for **GameObjects** and **scripts**. These icons display in the **Scene view**, along with built-in icons for items such as Lights and **Cameras**. Use the Gizmos menu to control how icons are drawn in the **Scene** view.

GameObject Select Icon button

To change the icon for a GameObject, select the GameObject in the Heirarchy window or Scene view, then click the **Select Icon** button (the blue cube, highlighted with a red square in the image below) in the **Inspector** window to the left of the GameObject's name.

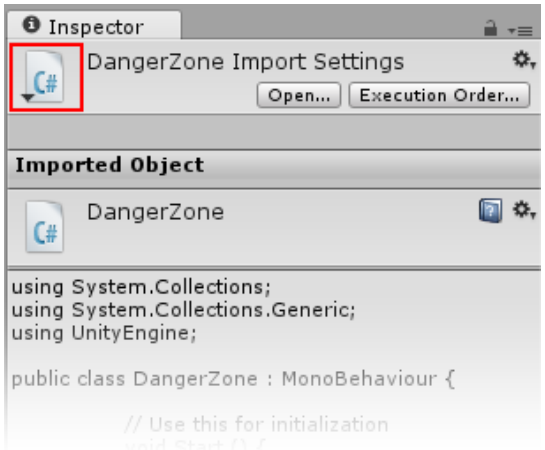


The GameObject **Select Icon** button (here highlighted with a red square) in the Inspector window

When you assign an icon to a GameObject, the icon displays in the Scene view over that GameObject (and any duplicates made afterwards). You can also assign the icon to a **Prefab** to apply the icon to all instances of that **Prefab** in the Scene.

Script Select Icon button

To assign a custom icon to a script, select the script in the Project window, then click the **Select Icon** button (the C# file icon, highlighted with a red square in the image below) in the Inspector window to the left of the script's name.

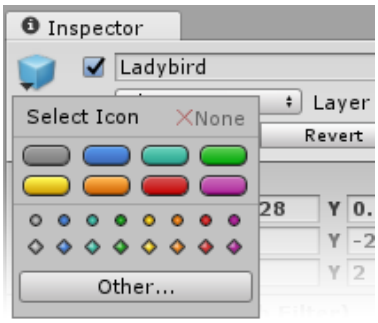


The Script **Select Icon** button (here highlighted with a red square) in the Inspector window

When you assign an icon to a script, the icon displays in the Scene view over any GameObject which has that script attached.

The Select Icon menu

Whether you are assigning an icon to a GameObject or a Script, the pop-up **Select Icon** menu is the same:

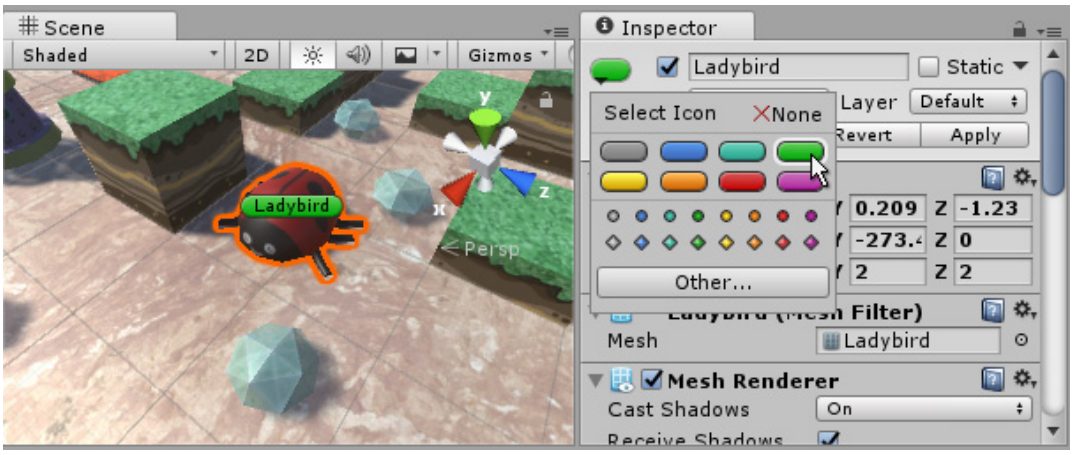


The **Select Icon** menu has built-in icons. Click on an icon to select it, or click **Other...** to select an image from your project **Assets** to use as the icon.

The built-in icons fall into two categories: **label icons** and **image-only icons**.

Label icons

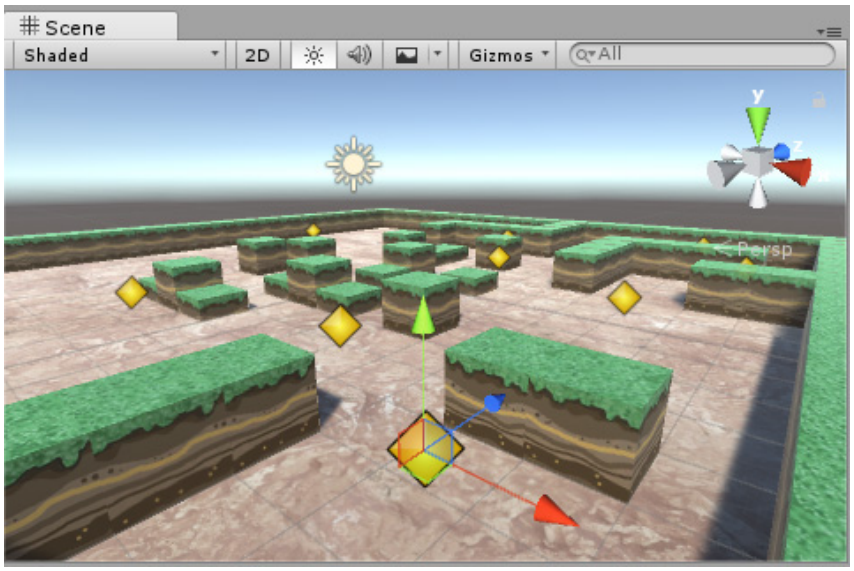
Assign a label icon to a GameObject or script to display the name of the GameObject in the Scene view.



A GameObject with a label icon assigned

Image-only icons

Image-only icons do not show the GameObject's name. These are useful for assigning to GameObjects that may not have a visual representation (for example, navigation waypoints). With an icon assigned, you can see and click on in it the Scene view to select and move an otherwise invisible GameObject.



A yellow diamond icon assigned to multiple invisible GameObjects

Any Asset image in your project can also be used as an icon. For example, a skull and crossbones icon could be used to indicate danger areas in your level.

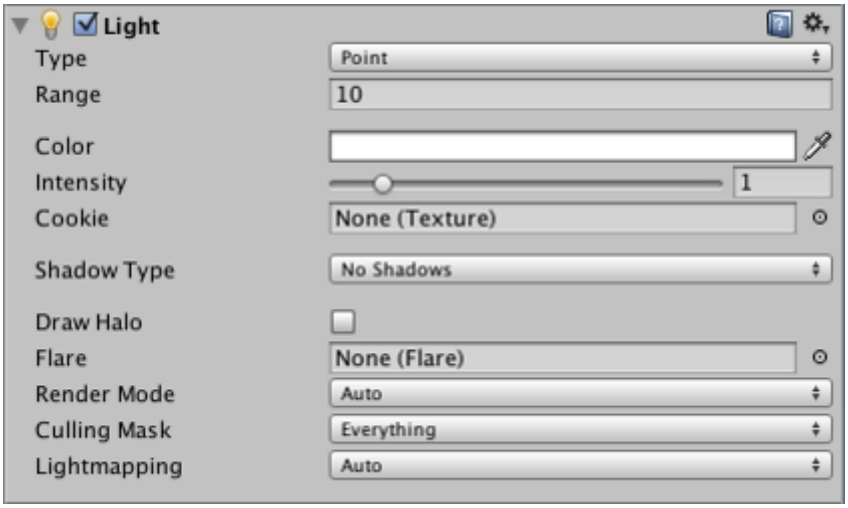
A skull and crossbones image has been assigned to a script. The icon is displayed over any GameObjects which have that script attached.

Note: When an Asset's icon is changed, the Asset itself is marked as modified and therefore picked up by **version control** systems.

Editing Properties

Leave feedback

Properties are settings and options for **components** that can be edited in the **inspector**.



Light component showing various value and reference properties

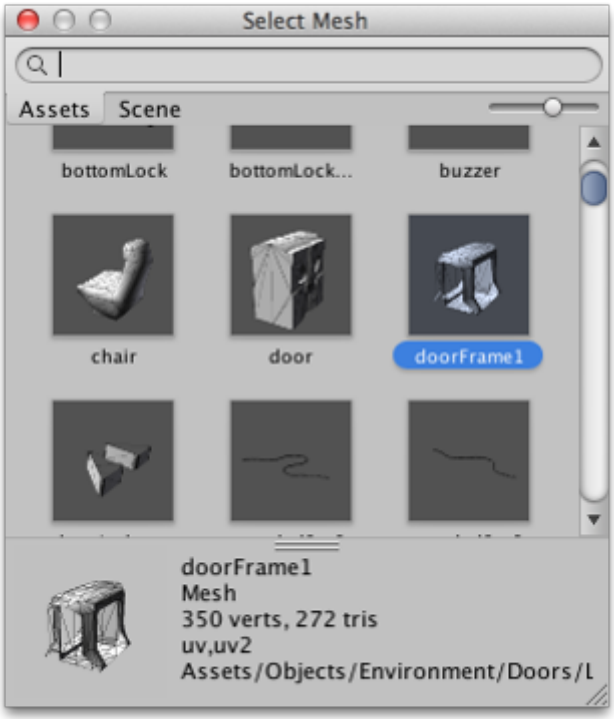
Properties can be broadly categorized as *references* (links to other objects and assets) or *values* (numbers, checkboxes, colors, etc).

References

References can be assigned by dragging and dropping an object or **asset** of the appropriate type to the property in the inspector. For example, the Mesh Filter component needs to refer to a Mesh asset somewhere in the project. When the component is initially created, the reference is unassigned:

...but you can assign a Mesh to it by dropping a Mesh asset onto it:

You can also use the **Object Picker** to select an object for a reference property. If you click the small circle icon to the right of the property in the inspector, you will see a window like this:



The object picker lets you search for and select objects within the **scene** or project assets (the information panel at the bottom of the window can be raised and lowered as desired). Choosing an object for the reference property is simply a matter of double clicking it in the picker.

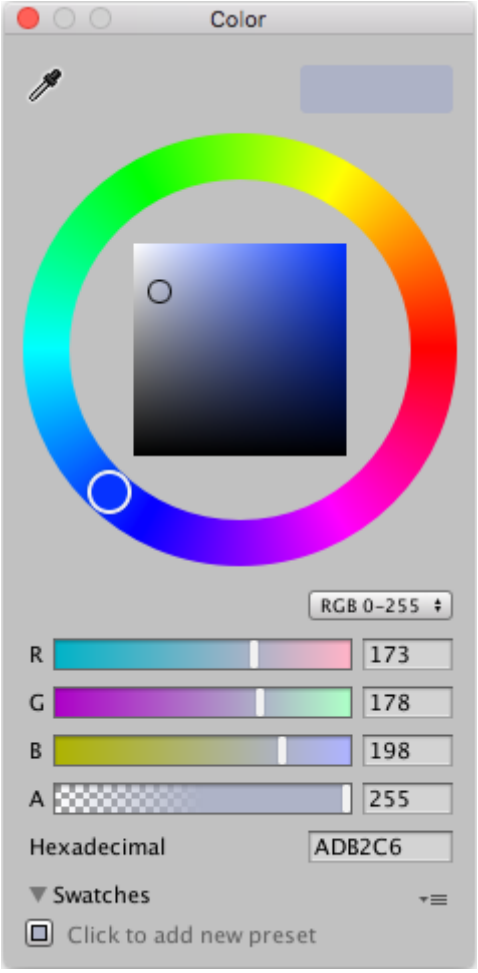
When a reference property is of a component type (such as Transform), you can drop any object onto it; Unity will locate the first component of that type on the object and assign it to the reference. If the object doesn't have any components of the right type, the assignment will be rejected.

Values

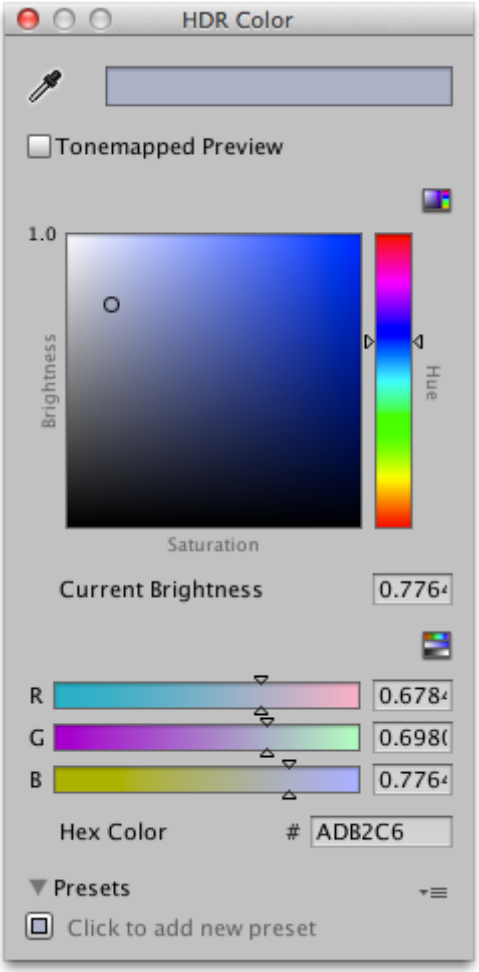
Most values are edited using familiar text boxes, checkboxes and menus, depending on their type (as a convenience, numeric values can also be moved up or down by dragging the mouse over the property name label). However, there are some values of more complex types that have their own specific editors. These are described below.

Colors

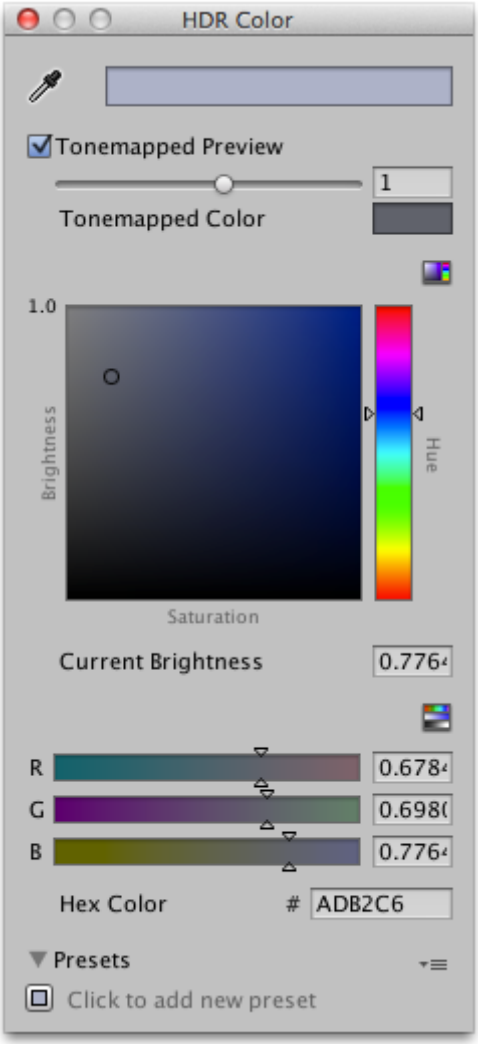
Color properties open the **Color Picker** when clicked.



The Unity Color Picker window



The HDR Color Picker window (displayed when clicking the emission color property in the standard shader or if you use the ColorUsageAttribute on a Color in your script)

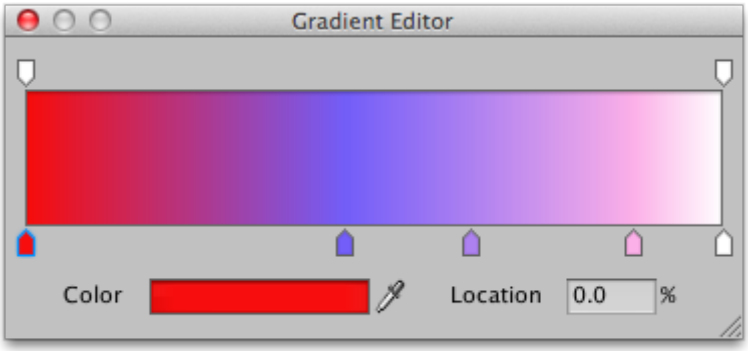


The HDR Color Picker window shown with the tonemapped option selected

Unity uses its own color picker but on Mac OS X you can choose to use the system picker from the Preferences (menu: **Unity > Preferences** and then **Use OS X Color Picker** from the *General* panel).

Gradients

In graphics and animation, it is often useful to be able to blend one colour gradually into another, over space or time. A **gradient** is a visual representation of a colour progression, which simply shows the main colours (which are called **stops**) and all the intermediate shades between them:



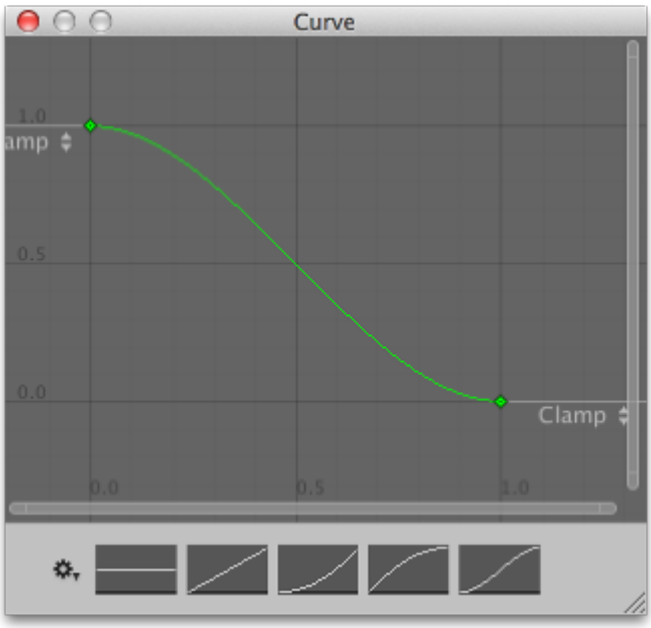
The upward-pointing arrows along the bottom of the gradient bar denote the stops. You can select a stop by clicking on it; its value will be shown in the Color box which will open the standard colour picker when clicked. A

new stop can be created by clicking just underneath the gradient bar. The position of any of the stops can be changed simply by clicking and dragging and a stop can be removed with **ctrl/cmd + delete**.

The downward-pointing arrows above the gradient bar are also stops but they correspond to the alpha (transparency) of the gradient at that point. By default, there are two stops set to 100% alpha (ie, fully opaque) but any number of stops can be added and edited in much the same way as the colour stops.

Curves

A **Curve** is a line graph that shows the response (on the Y axis) to the varying value of an input (on the X axis).

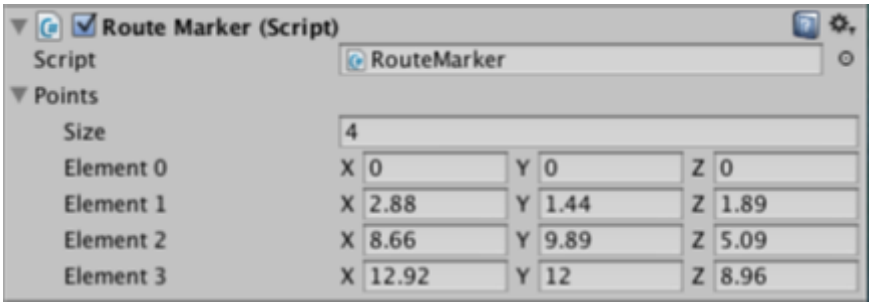


Curve editor

Curves are used in a variety of different contexts within Unity, especially in animation, and have a number of different options and tools. These are explained on the Editing Curves page of the manual.

Arrays

When a script exposes an array as a public variable, the inspector will show a value editor that lets you edit both the size of the array and the values or references within it.

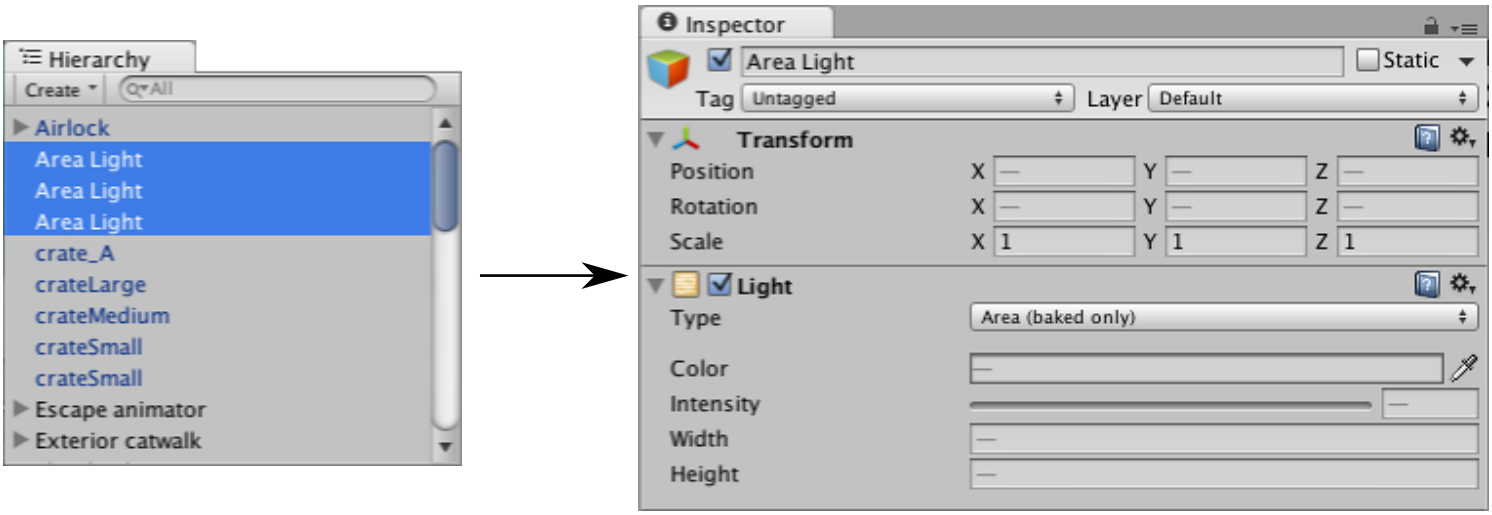


Script with a Vector3 array property

When you decrease the *Size* property, values from the end of the array will be removed. When you increase the size, the current last value will be copied into all the new elements added. This can be useful when setting up an array whose values are mostly the same - you can set the first element and then change the size to copy it to all the other elements.

Editing Several Objects at Once

When you have two or more objects selected, the components they have in common can all be edited together (ie, the values you supply will be copied to all the selected objects).



Inspector showing a multiple selection

Where property values are the same for all selected objects, the value will be shown but otherwise, it will be shown as a dash character. Only components that are common to *all* objects will be visible in the inspector. If any selected object has components that are not present on other objects, the inspector will show a message to say that some components are hidden. The context menu for a property (opened by right-clicking on its name label) also has options to allow you to set its value from any of the selected components.

Note that you can also edit several selected instances of a **prefab** at once but the usual *Select*, *Revert* and *Apply* buttons will not be available.

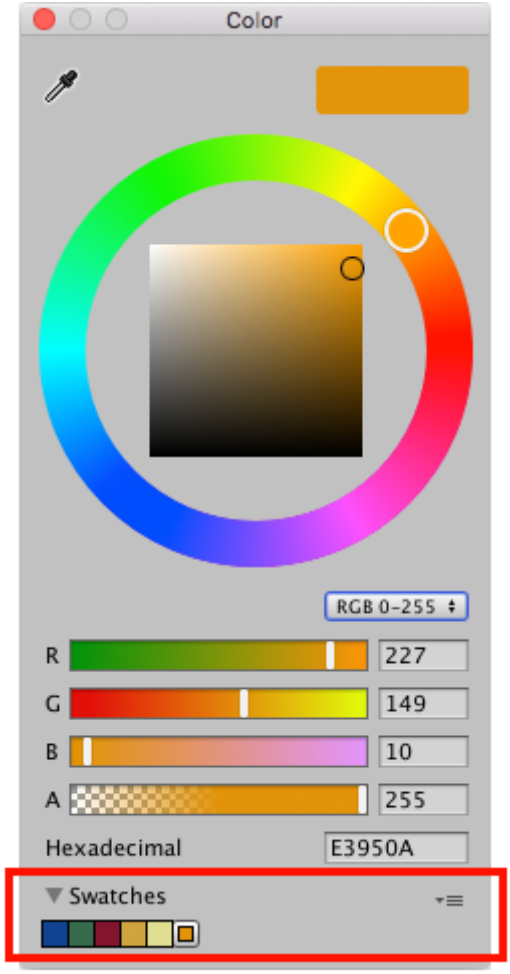
2018–04–19 Page amended with limited editorial review

Preset libraries renamed to swatch libraries in 2018.1

Swatch libraries

Leave feedback

Use swatch libraries to reuse, save, and share colors, gradients, and **animation curves**. You can save and choose swatches in the Color Picker, Gradient Editor, and Curve Editor.



Swatches section (red) in the Unity Color Picker

A swatch library is a collection of swatches that you save in a file. The Swatches section displays a single library at a time.

To save a swatch:

Open the Color Picker, Gradient Editor, or Curve Editor. For example, select **Main Camera** in the Hierarchy window.
In the Inspector window, click **Background Color**.
In the picker window, adjust the color, gradient, or curve to your liking.
In **Swatches**, click the outlined box.
If the view is in List mode, optionally type a name for the swatch.

Example of saving a color while in the Grid view

Drag and drop swatches to change their order. Right-click a swatch to move it to the top, replace it, rename it, or delete it. You can also delete a swatch by Alt/Option-clicking it.

Use the drop-down menu in **Swatches** to:

Choose **List** or **Grid** to change the view. The List view also displays the names of swatches.
Choose a swatch library.
Choose **Create a Library** to create a new swatch library and the location to save it in.
Choose **Reveal Current Library Location** to view the current library in Windows Explorer/Mac OS Finder.

By default, Unity saves swatch libraries in your user preferences. You can also save a swatch library in your Project. Unity saves Project swatch libraries in the *Editor* folder of the *Assets* folder. To share Project swatch libraries between users, or to include them in a package, add them to a revision control repository.

To edit a Project swatch library:

Select the swatch library in the Project window.
In the Inspector window, click **Edit**.

2018–04–19 Page amended with editorial review

Preset libraries renamed to swatch libraries in 2018.1

Inspector Options

Leave feedback

The **Inspector** Lock and the **Inspector** Debug Mode are two useful options that can help you in your workflow.

Lock

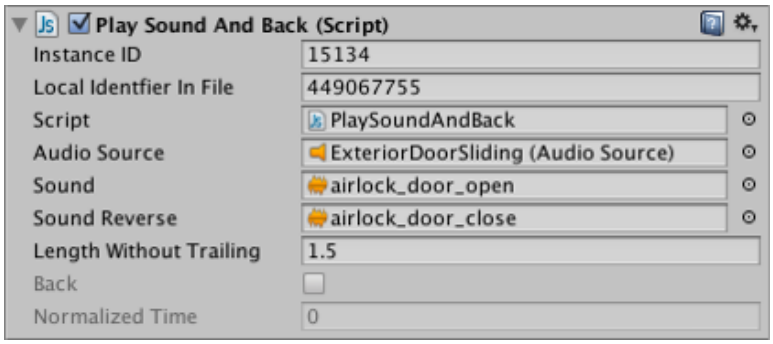
Normally, the inspector shows the details of the currently selected object but it is sometimes useful to keep one object in the inspector while you work with others. To enable this, the inspector has a *Lock* mode that you can activate with the small padlock icon to the top right of the inspector window.

Note that you can have more than one inspector open at once (menu: **Add Tab** from the inspector tab menu), so you could keep one locked and have a second one to show the current selection. Below is an example of adding a new tab and locking it so it retains the desired view (in this case, the Time Manager editor settings). The result is two inspector windows, both visible, each showing different content.

Adding a new tab, docking it under the current inspector tab, and locking it.

Debug/Normal Mode

Another option on the tab menu is the choice between *Normal* mode and *Debug* mode. Normally, the inspector shows a custom editor for an **asset** or **component** if one is available but sometimes it is handy to see the properties directly. Debug mode shows just the properties rather than a custom editor and for **scripts**, it also displays private variables (although their values can't be edited like those of public variables).

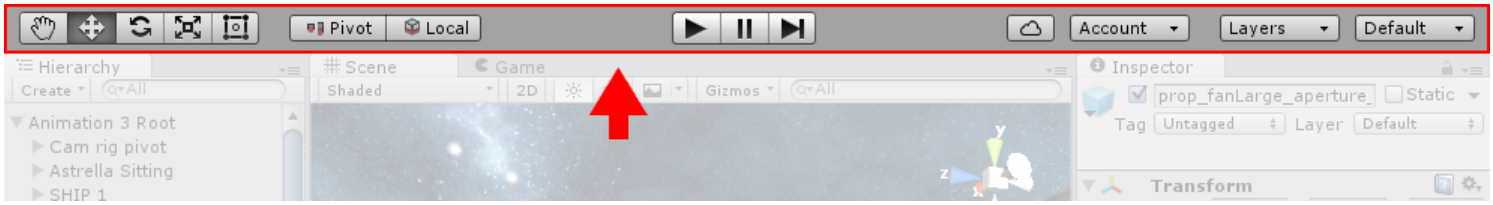


Script as seen in Debug mode with greyed private variables

As with Lock mode, Debug/Normal mode is applied to each inspector individually, so you can have two inspectors open at once to see both views.

The Toolbar

Leave feedback



The **Toolbar** consists of seven basic controls. Each relate to different parts of the Editor.

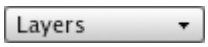
Transform Tools – used with the Scene View

Transform **Gizmo** Toggles – affect the Scene View display

Play/Pause/Step Buttons – used with the Game View

Cloud Button - opens the Unity Services Window.

Account Drop-down - used to access your Unity Account.



Layers Drop-down – controls which objects are displayed in **Scene** View

Layout Drop-down – controls arrangement of all Views

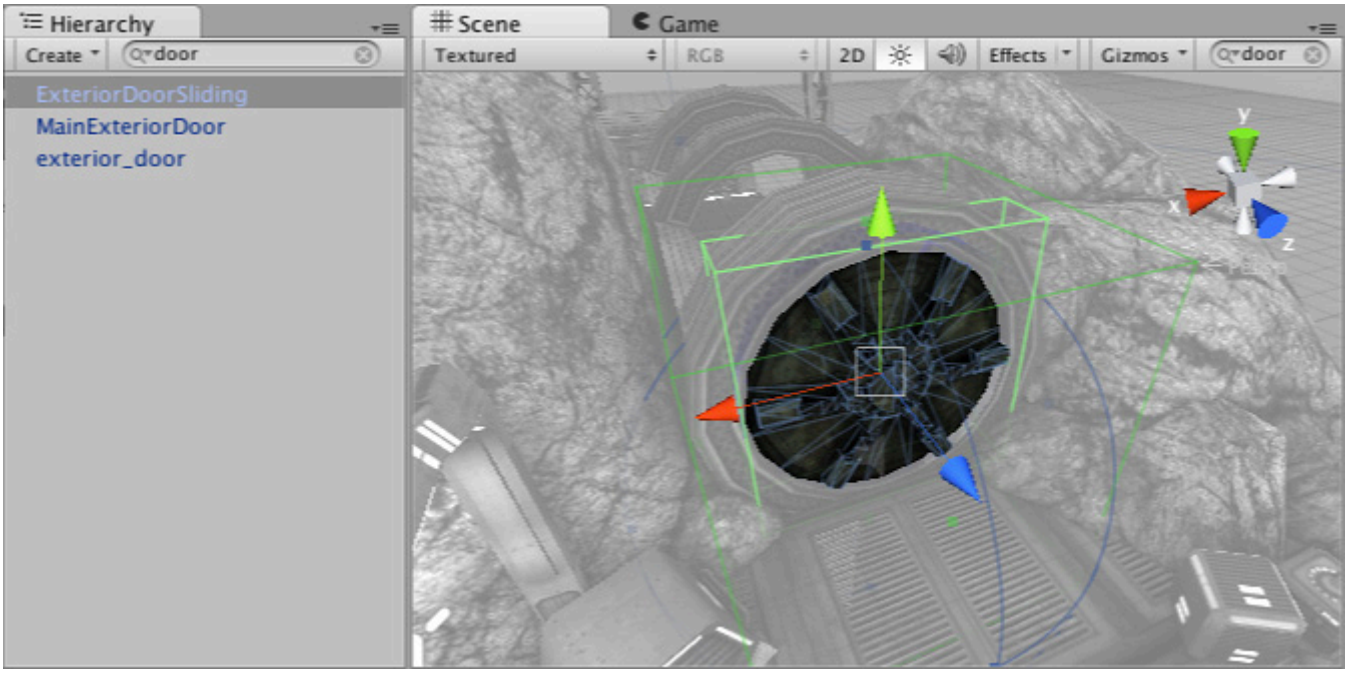
Searching

Leave feedback

When working with large complex **scenes** it can be useful to search for specific objects. By using the **Search** feature in Unity, you can filter out only the object or group of objects that you want to see. You can search assets by their name, by **Component** type, and in some cases by **asset** *Labels* (see below). You can specify the search mode by choosing from the Search drop-down menu.

Scene Search

Both the Scene and Hierarchy views have a search box that allows you to filter objects by their names. Since the two views are basically just different representations of the same set of objects, any search query you type will be duplicated in both search boxes and applied to both views at the same time. Note that both views change slightly when a search is active: the **Scene View** will show filtered-out objects in grey and the Hierarchy view will lose the hierarchic information and simply show objects listed by name:



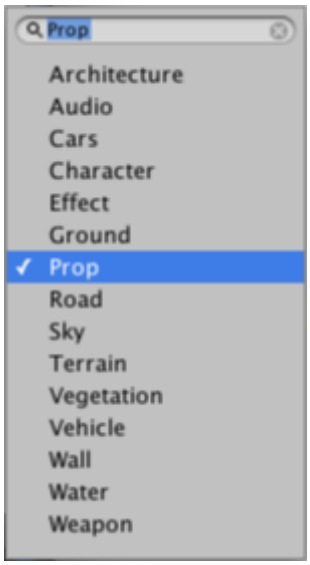
Scene and Hierarchy views with search filtering applied.

The small cross button at the right of the search box removes the search query and restores the view to normal. The menu at the left hand side of the box lets you choose whether to filter objects by name, by type or both at once.

Project Search and Labels

There is also a search box in the **Project view**. The search applies to *assets* rather than their instances in the scene.

One additional option available for assets is to search by **Labels** as well as by name and type. A label is simply a short piece of text that you can use to group particular assets. If you click on the label button second from the left of the Project window you will see a menu of existing labels with a text box at the top. You can then request assets assigned specific label. Also you can create new labels. For example you could add a “Vehicles” label to make all vehicle assets easy to locate. You can add a label to an asset from the *Asset Labels* box at the bottom of its inspector.



Adding a “Prop” label

The text box lets you filter the existing labels or enter the text of a new label; you can press the spacebar or enter key while typing to add the new label text to the asset. Labels currently applied to an asset are shown with a check mark to their left in the menu. You can simply “select” an applied label from the menu to remove it. Note that any asset can have as many labels as desired and thereby belong to several different label groups at once.

See Inspector manual page for more informatation.

Other Windows

Leave feedback

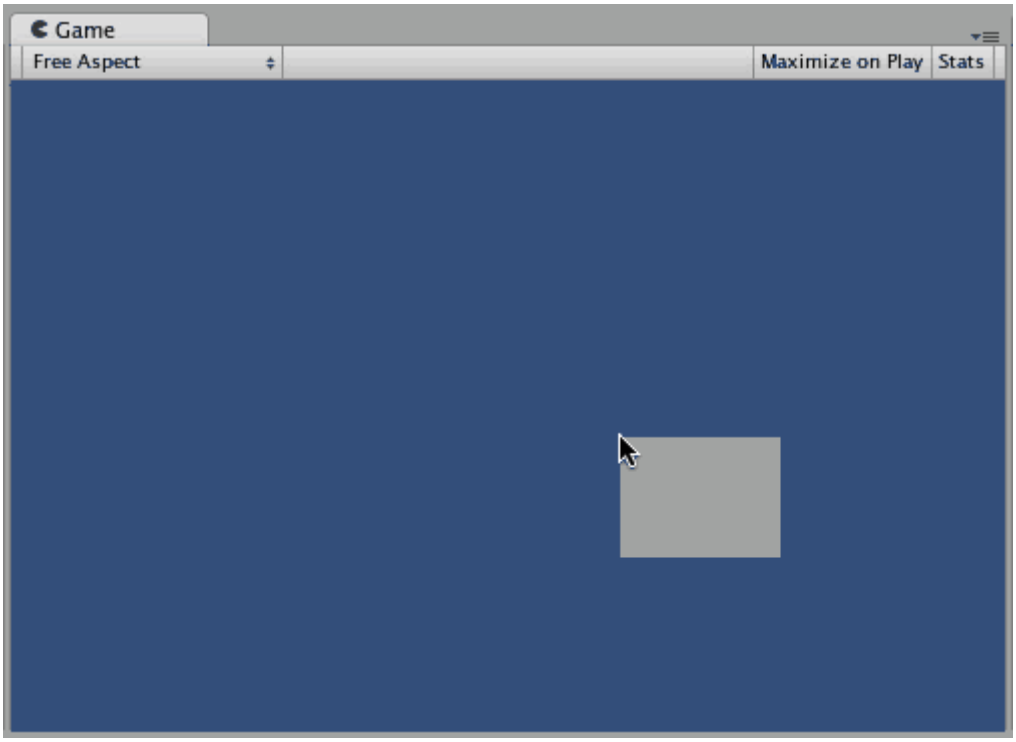
The Windows described on this page covers the basics of the interface in Unity. The other Windows in Unity are described elsewhere on separate pages:

The Console Window shows logs of messages, warnings, and errors.
The Animation Window can be used to animate objects in the **scene**.
The Profiler Window can be used to investigate and find the performance bottle-necks in your game.
The Lighting Window can be used to manage the lighting in your scene.
The Occlusion Culling Window can be used to manage **Occlusion Culling** for improved performance.

Customizing Your Workspace

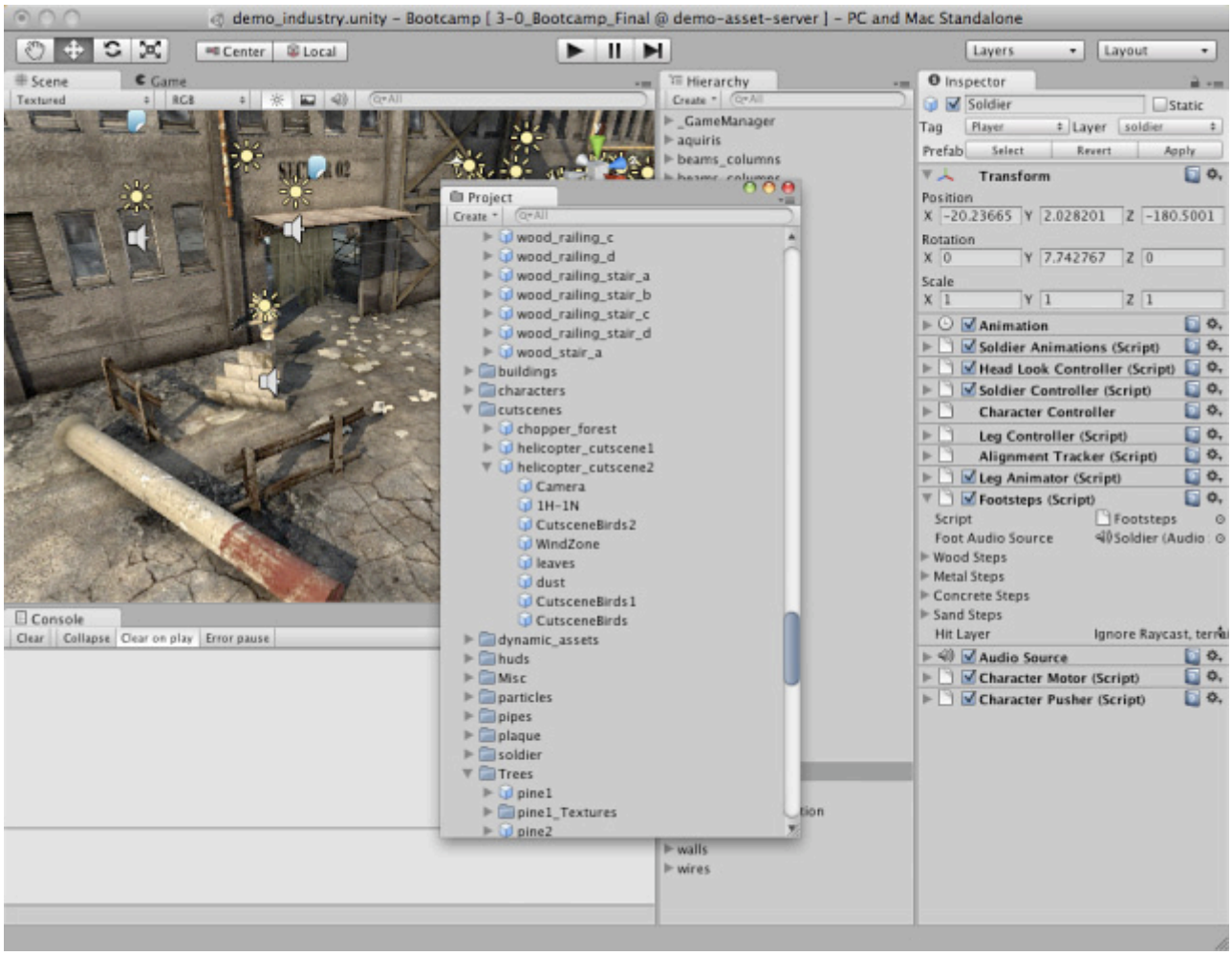
Leave feedback

You can customize your **Layout** of Views by click-dragging the Tab of any View to one of several locations. Dropping a Tab in the **Tab Area** of an existing window will add the Tab beside any existing Tabs. Alternatively, dropping a Tab in any **Dock Zone** will add the View in a new window.



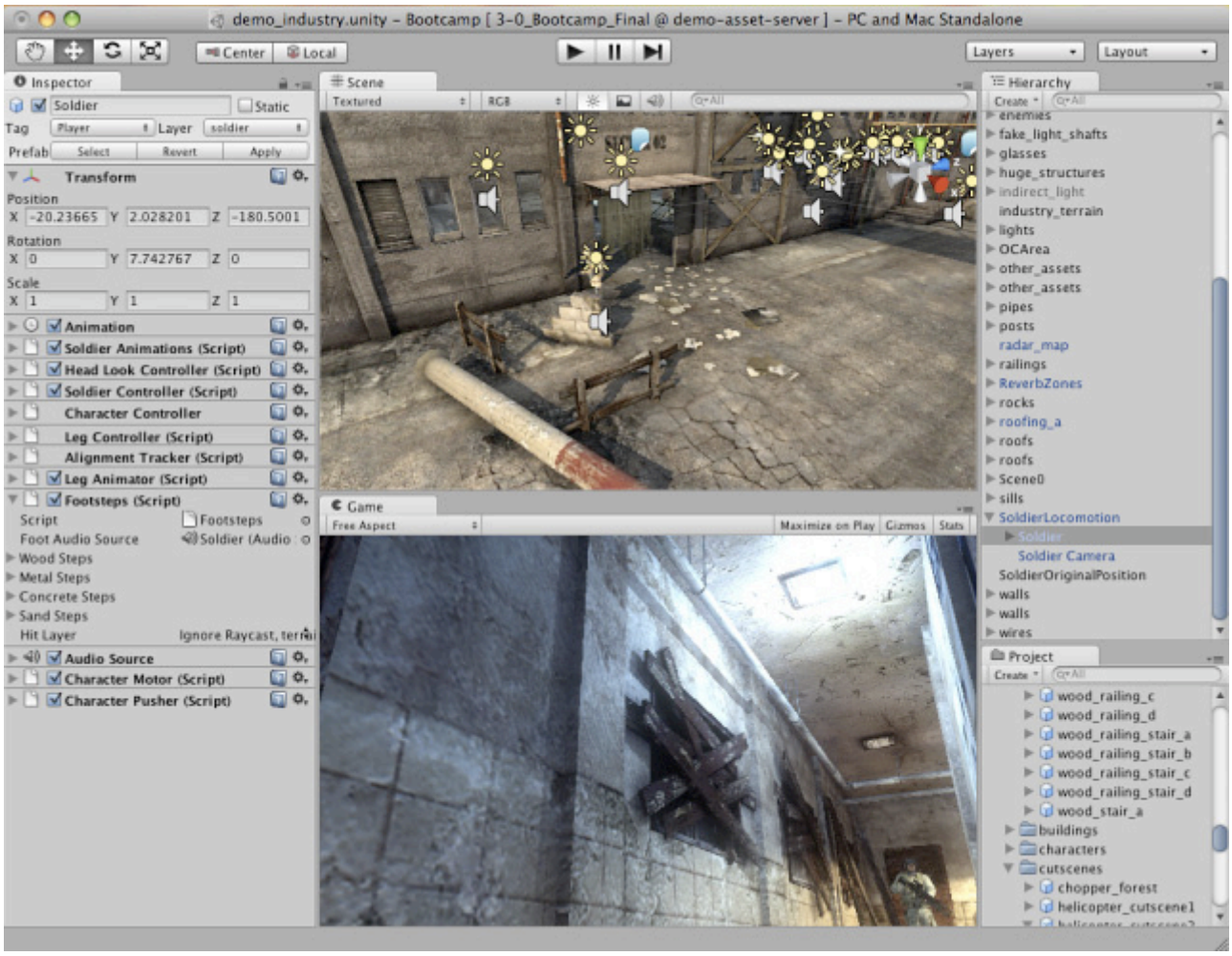
Views can be docked to the sides or bottom of any existing window

Tabs can also be detached from the Main Editor Window and arranged into their own floating Editor Windows. Floating Windows can contain arrangements of Views and Tabs just like the Main Editor Window.



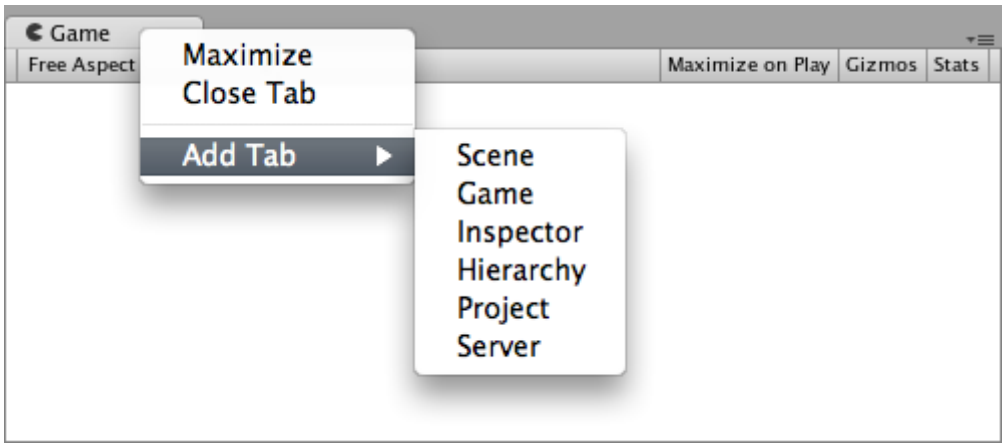
Floating Editor Windows are the same as the Main Editor Window, except there is no Toolbar

When you've created a Layout of Editor Windows, you can Save the layout and restore it any time. You do this by expanding the Layout drop-down (found on the Toolbar) and choosing **Save Layout...**. Name your new layout and save it, then restore it by simply choosing it from the Layout drop-down.



A completely custom Layout

At any time, you can right-click the tab of any view to view additional options like Maximize or add a new tab to the same window.



Unity hotkeys

Leave feedback

This page gives an overview of the default Unity keyboard shortcuts. You can also download a PDF of the table for Windows and MacOSX. Where a command has **Ctrl/Cmd** as part of the keystroke, this indicates that the **Control** key should be used on Windows and the **Command** key on MacOSX.

The Mac trackpad also has a number of shortcuts for navigating the **Scene view**. See Scene view navigation to learn more about these.

Tools

Keystroke	*Command*
Q	Pan
W	Move
E	Rotate
R	Scale
T	Rect Tool
Z	Pivot Mode toggle
X	Pivot Rotation Toggle
V	Vertex Snap
CTRL/CMD+LMB	Snap

GameObject

Ctrl/Cmd+Shift+N	New empty game object
Alt+Shift+N	New empty child to selected game object
Ctrl/Cmd+Alt+F	Move to view
Ctrl/Cmd+Shift+F	Align with view
Shift+F or double-F	Locks the **scene** view **camera** to the selected GameObject

Window

Ctrl/Cmd+1	Scene
Ctrl/Cmd+2	Game
Ctrl/Cmd+3	**Inspector**
Ctrl/Cmd+4	Hierarchy
Ctrl/Cmd+5	Project
Ctrl/Cmd+6	Animation
Ctrl/Cmd+7	**Profiler**
Ctrl/Cmd+9	**Asset store**
Ctrl/Cmd+0	**Version Control**
Ctrl/Cmd+Shift+C	Console

Edit

Ctrl/Cmd+Z	Undo
Ctrl+Y (Windows only)	Redo
Cmd+Shift+Z (Mac only)	Redo
Ctrl/Cmd+X	Cut
Ctrl/Cmd+C	Copy
Ctrl/Cmd+V	Paste
Ctrl/Cmd+D	Duplicate
Shift+Del	Delete
F	Frame (centre) selection
Ctrl/Cmd+F	Find
Ctrl/Cmd+A	Select All
Ctrl/Cmd+P	Play

Ctrl/Cmd+Shift+P	Pause
Ctrl/Cmd+Alt+P	Step

Selection

Ctrl/Cmd+Shift+1	Load Selection 1
Ctrl/Cmd+Shift+2	Load Selection 2
Ctrl/Cmd+Shift+3	Load Selection 3
Ctrl/Cmd+Shift+4	Load Selection 4
Ctrl/Cmd+Shift+5	Load Selection 5
Ctrl/Cmd+Shift+6	Load Selection 6
Ctrl/Cmd+Shift+7	Load Selection 7
Ctrl/Cmd+Shift+8	Load Selection 8
Ctrl/Cmd+Shift+9	Load Selection 9
Ctrl/Cmd+Alt+1	Save Selection 1
Ctrl/Cmd+Alt+2	Save Selection 2
Ctrl/Cmd+Alt+3	Save Selection 3
Ctrl/Cmd+Alt+4	Save Selection 4
Ctrl/Cmd+Alt+5	Save Selection 5
Ctrl/Cmd+Alt+6	Save Selection 6
Ctrl/Cmd+Alt+7	Save Selection 7
Ctrl/Cmd+Alt+8	Save Selection 8
Ctrl/Cmd+Alt+9	Save Selection 9

Assets

Ctrl/Cmd+R	Refresh

Note: The following Animation hotkeys only work in the Animation window.

Animation

Shift+Comma	First **Keyframe**
Shift+K	Key Modified
K	Key Selected
Shift+Period	Last Keyframe
Period	Next Frame
Alt+Period	Next Keyframe
Space	Play Animation
Comma	Previous Frame
Alt+Comma	Previous Keyframe

2017–11–24 Page amended with limited editorial review

Creating Gameplay

Leave feedback

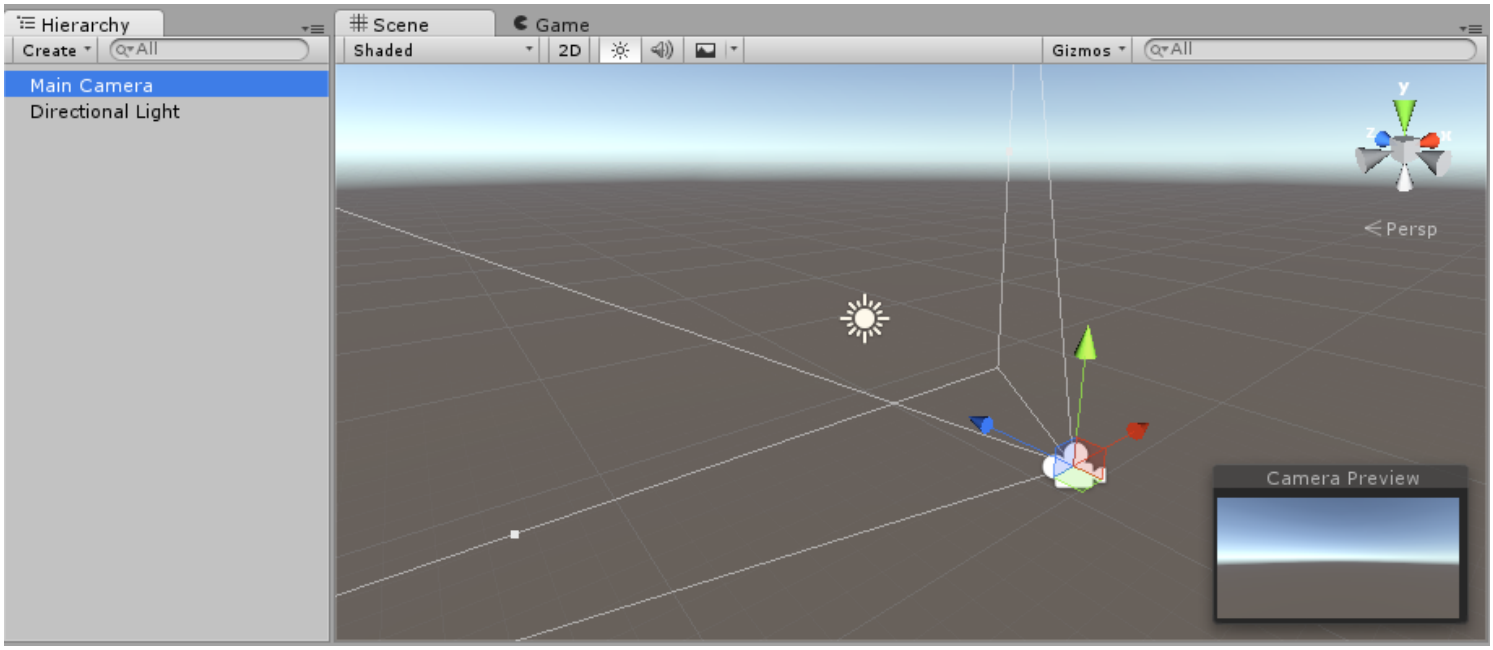
Unity empowers game designers to make games. What's really special about Unity is that you don't need years of experience with code or a degree in art to make fun games. There are a handful of basic workflow concepts needed to learn Unity. Once understood, you will find yourself making games in no time. With the time you will save getting your games up and running, you will have that much more time to refine, balance, and tweak your game to perfection.

This section will explain the core concepts you need to know for creating gameplay mechanics. The majority of these concepts require you to write **Scripts**. For an overview of creating and working with Scripts, read the Scripting page.

Scenes

Leave feedback

Scenes contain the environments and menus of your game. Think of each unique **Scene** file as a unique level. In each **Scene**, you place your environments, obstacles, and decorations, essentially designing and building your game in pieces.



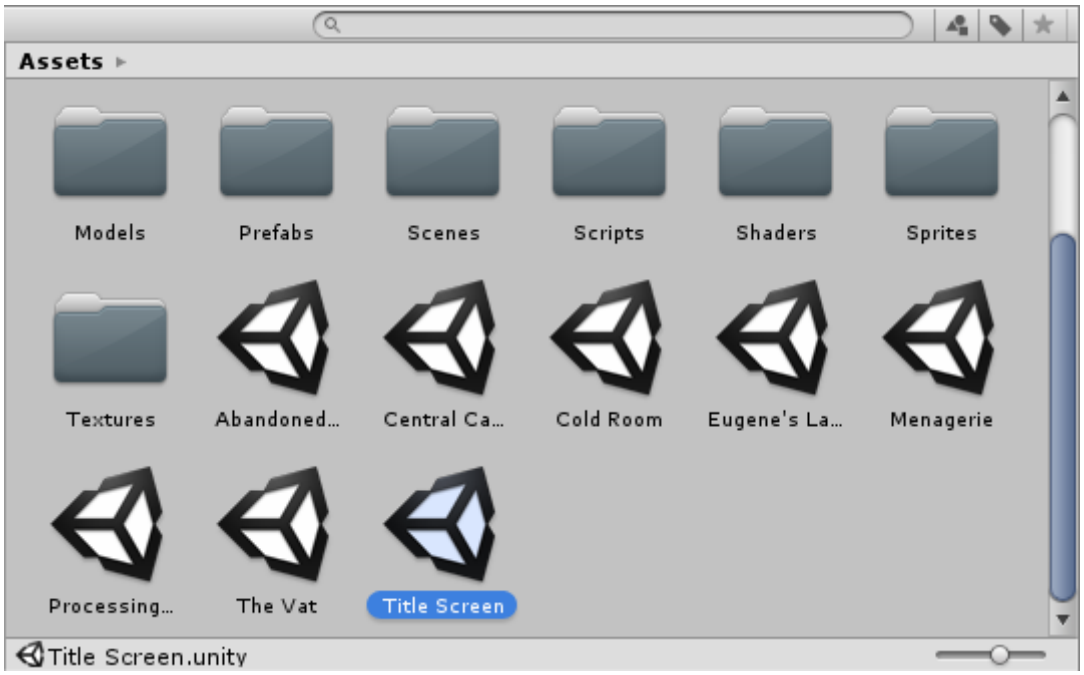
A new empty Scene, with the default 3D objects: a Main Camera and a directional Light

When you create a new Unity project, your **scene view** displays a new Scene. This Scene is *untitled* and *unsaved*. The Scene is empty except for a **Camera** (called **Main Camera**) and a Light (called **Directional Light**).

Saving Scenes

To save the Scene you're currently working on, choose **File** > **Save Scene** from the menu, or press Ctrl + S (Windows) or Cmd + S (masOS).

Unity saves Scenes as **Assets** in your project's *Assets* folder. This means they appear in the Project window, with the rest of your **Assets**.



Saved Scene Assets visible in the Project window

Opening Scenes

To open a Scene in Unity, double-click the Scene Asset in the Project window. You must open a Scene in Unity to work on it.

If your current Scene contains unsaved changes, Unity asks you whether you want to save or discard the changes.

Multi-Scene editing

It is possible to have multiple Scenes open for editing at one time. For more information about this, see documentation on Multi-Scene editing.

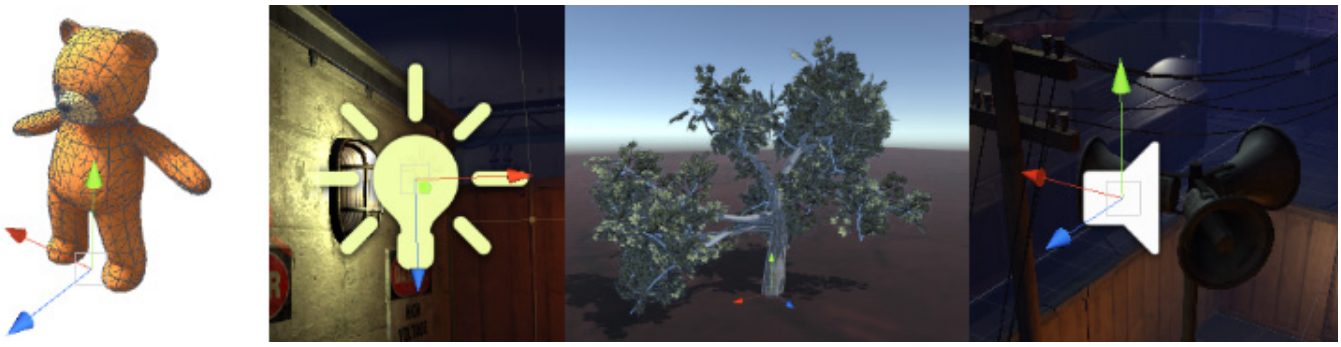
2017–08–01 Page amended with limited editorial review

GameObjects

Leave feedback

The **GameObject** is the most important concept in the Unity Editor.

Every object in your game is a **GameObject**, from characters and collectible items to lights, **cameras** and special effects. However, a GameObject can't do anything on its own; you need to give it properties before it can become a character, an environment, or a special effect.



Four different types of GameObject: an animated character, a light, a tree, and an audio source

To give a GameObject the properties it needs to become a light, or a tree, or a camera, you need to add components to it. Depending on what kind of object you want to create, you add different combinations of components to a GameObject.

You can think of a GameObject as an empty cooking pot, and components as different ingredients that make up the recipe of your game. Unity has lots of different built-in component types, and you can also make your own components using the Unity Scripting API.

This section explains how GameObjects, components and the Scripting API fit together, and how to create and use them.

2017–08–01 Page amended with limited editorial review

GameObject

Leave feedback

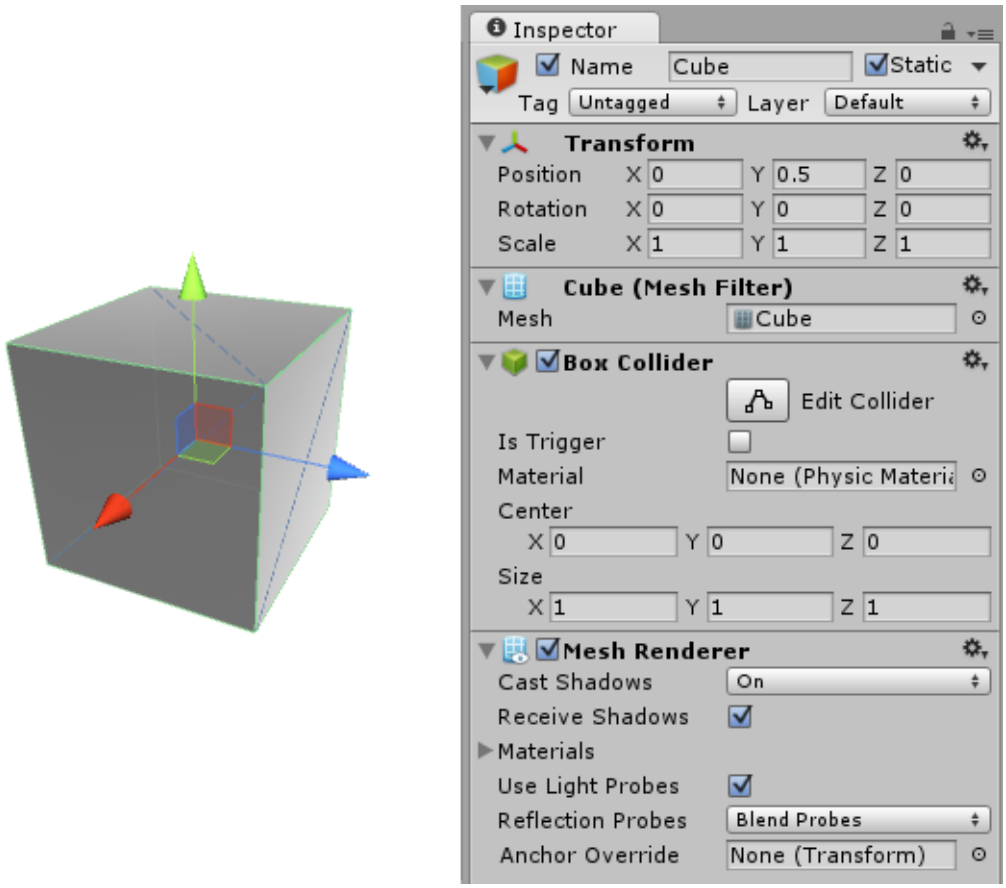
SWITCH TO SCRIPTING

GameObjects are the fundamental objects in Unity that represent characters, props and scenery. They do not accomplish much in themselves but they act as containers for **Components**, which implement the real functionality.

For example, a Light object is created by attaching a Light component to a GameObject.

A simple Cube GameObject with several Components

A solid cube object has a **Mesh Filter** and **Mesh** Renderer component, to draw the surface of the cube, and a Box **Collider** component to represent the object's solid volume in terms of physics.



A simple Cube GameObject with several Components

Details

A GameObject always has a Transform component attached (to represent position and orientation) and it is not possible to remove this. The other components that give the object its functionality can be added from the editor's **Component** menu or from a script. There are also many useful pre-constructed objects (primitive shapes, Cameras, etc) available on the **GameObject > 3D Object** menu, see Primitive Objects.

Since GameObjects are a very important part of Unity, there is a GameObjects section in the manual with extensive detail about them. You can find out more about controlling GameObjects from **scripts** on the GameObject scripting reference page.

Introduction to components

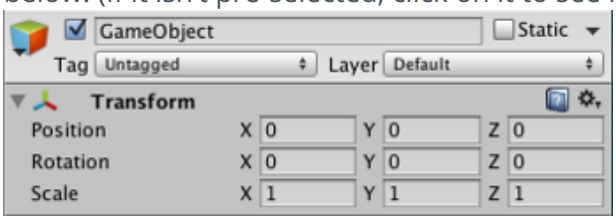
Leave feedback

A **GameObject** contains **components**. (See documentation on GameObject for more information.)

Below is an example of how the **GameObject** and **component** relationship works using the most common **component**, the **Transform Component**.

You can see the **Transform Component** by looking at the **Inspector** for a new **GameObject**:

Open any scene in any project in the Unity Editor. (See documentation on Getting Started for guidance on this.)
Create a new **GameObject** (menu: **GameObject** > **Create Empty**).
The new **GameObject** is pre-selected, with the **Inspector** showing its **Transform Component**, as in the image below. (If it isn't pre-selected, click on it to see its **Inspector**.)



The Inspector of a new, empty GameObject showing the Transform Component

Notice that the new, empty **GameObject** contains a name ("GameObject"), a Tag ("Untagged"), and a Layer ("Default"). It also contains a **Transform Component**.

The Transform Component

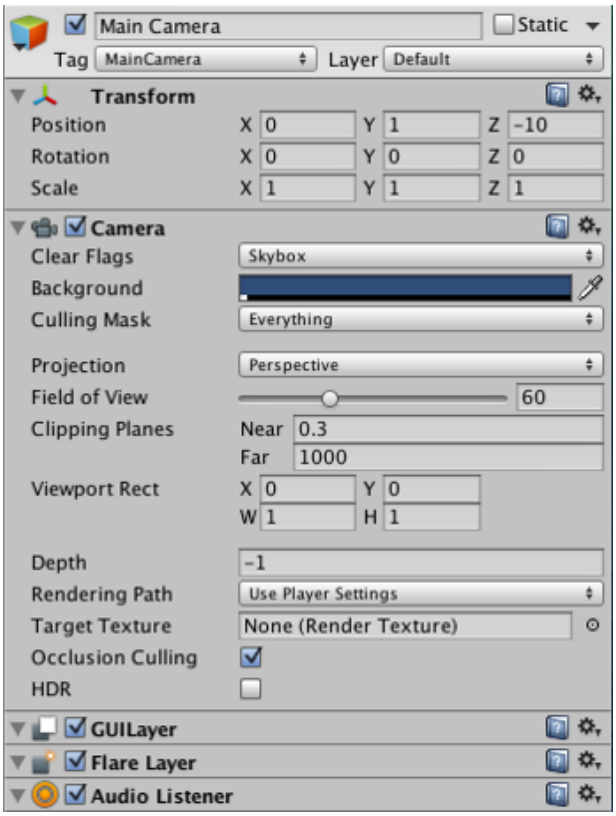
It is impossible to create a **GameObject** in the Editor without a **Transform Component**. This component defines the **GameObject's** position, rotation, and scale in the game world and **Scene view**.

The **Transform Component** also enables a concept called 'parenting' which is a critical part of working with **GameObjects**. To learn more about the **Transform Component** and parenting, see the Transform Component Reference page.

Other components

The **Transform Component** is critical to all **GameObjects**, so each **GameObject** has one but **GameObjects** can contain other **components** as well.

Every Scene has a **Main Camera GameObject** by default. It has several **components**.(You can see see this by selecting it in your open **Scene** to view its **Inspector**.)



The Main Camera is a type of GameObject - it is in each scene by default and has several components by default

Looking at the **Inspector** of the **Main Camera GameObject**, you can see that it contains additional **components**. Specifically, a Camera Component, a GUILayer, a Flare Layer, and an Audio Listener. All of these **components** provide functionality to this **GameObject**.

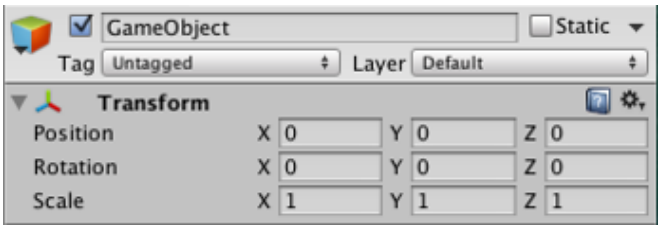
Rigidbody, Collider, Particle System, and Audio are all different **components** that you can add to a **GameObject**.

Using Components

Leave feedback

Components are the nuts & bolts of objects and behaviors in a game. They are the functional pieces of every **GameObject**. If you don't yet understand the relationship between Components and GameObjects, read the GameObjects page before going any further.

A GameObject is a container for many different Components. By default, all GameObjects automatically have a **Transform** Component. This is because the Transform dictates where the GameObject is located, and how it is rotated and scaled. Without a Transform Component, the GameObject wouldn't have a location in the world. Try creating an empty GameObject now as an example. Click the **GameObject->Create Empty** menu item. Select the new GameObject, and look at the **Inspector**.

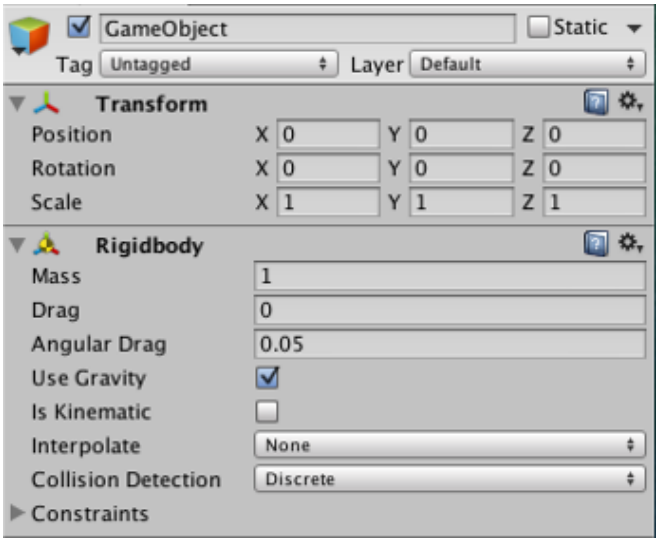


Even empty GameObjects have a Transform Component

Remember that you can always use the Inspector to see which Components are attached to the selected GameObject. As Components are added and removed, the Inspector will always show you which ones are currently attached. You will use the Inspector to change all the properties of any Component (including scripts).

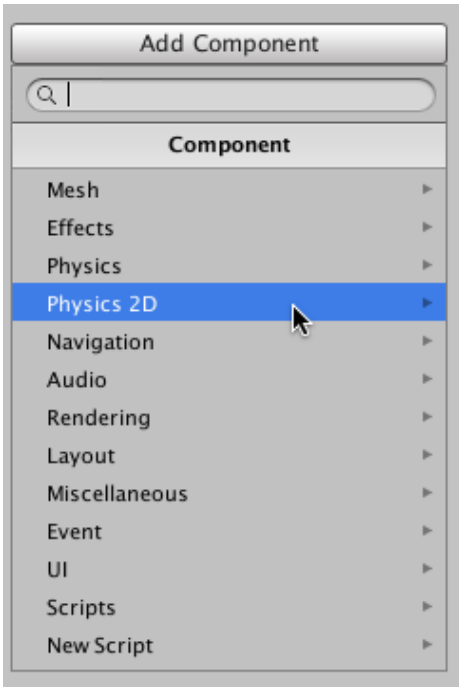
Adding Components

You can add Components to the selected GameObject through the Components menu. We'll try this now by adding a **Rigidbody** to the empty GameObject we just created. Select it and choose **Component->Physics->Rigidbody** from the menu. When you do, you will see the Rigidbody's properties appear in the Inspector. If you press **Play** while the empty GameObject is still selected, you might get a little surprise. Try it and notice how the Rigidbody has added functionality to the otherwise empty GameObject. (The Y position of the GameObject's transform starts to decrease. This is because the physics engine in Unity is causing the GameObject to fall under gravity.)



An empty GameObject with a Rigidbody Component attached

Another option is to use the **Component Browser**, which can be activated with the **Add Component** button in the object's inspector.



The Component Browser

The browser lets you navigate the components conveniently by category and also has a search box that you can use to locate components by name.

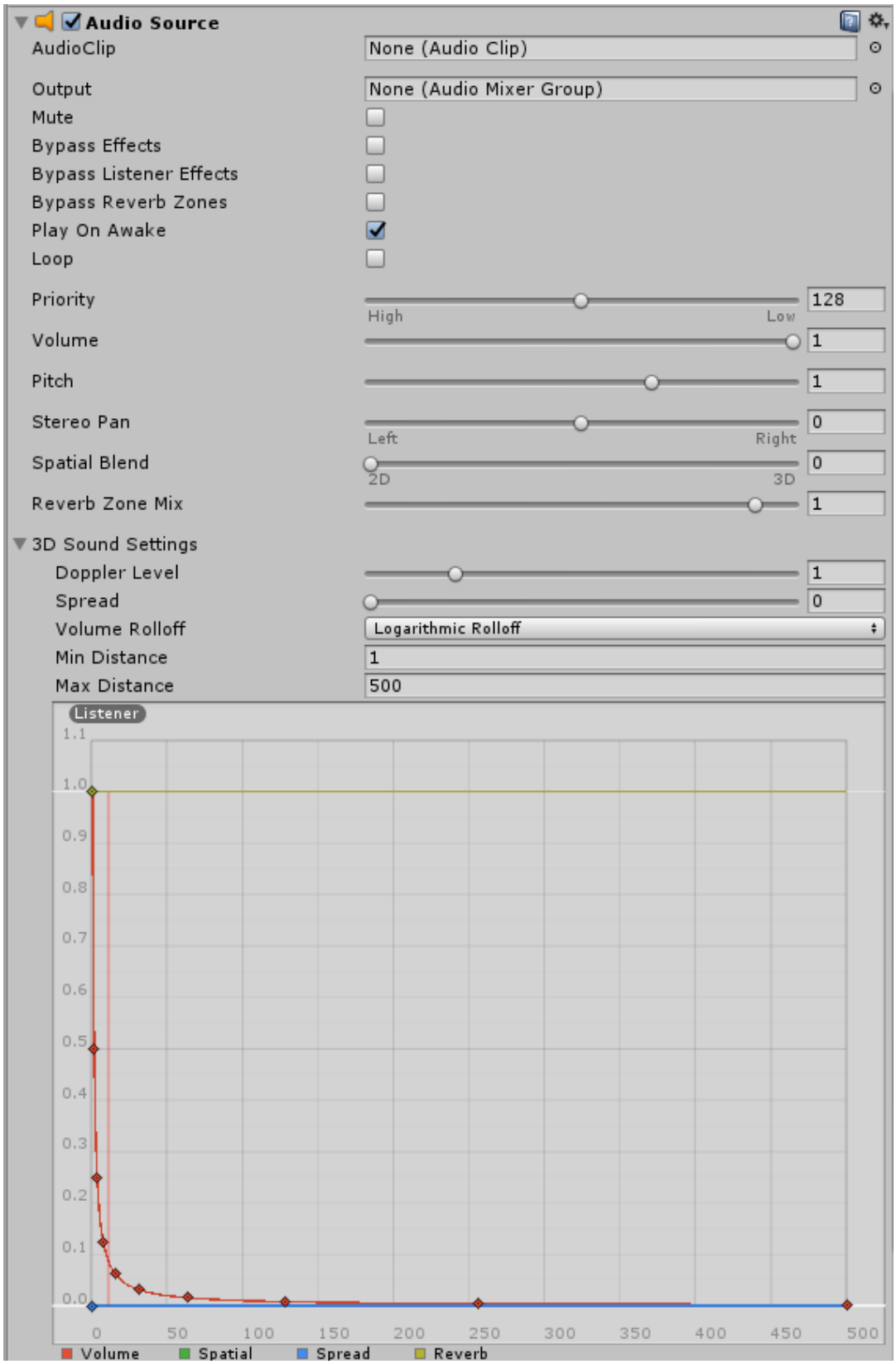
You can attach any number or combination of Components to a single GameObject. Some Components work best in combination with others. For example, the Rigidbody works with any **Collider**. The Rigidbody controls the Transform through the **NVIDIA PhysX physics engine**, and the **Collider** allows the Rigidbody to collide and interact with other Colliders.

If you want to know more about using a particular Component, you can read about any of them in the relevant Component Reference page. You can also access the reference page for a Component from Unity by clicking on the small **?** on the Component's header in the Inspector.

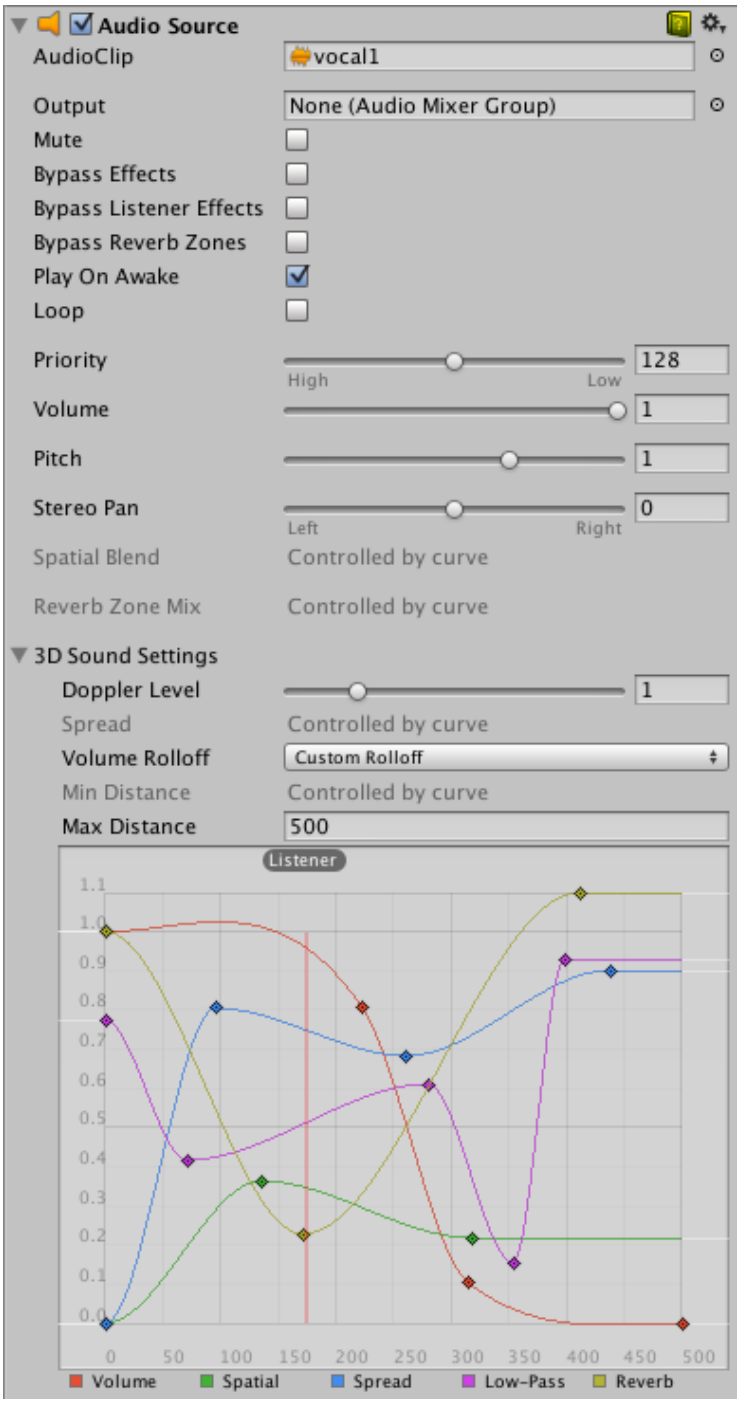
Editing Components

One of the great aspects of Components is flexibility. When you attach a Component to a GameObject, there are different values or **Properties** in the Component that can be adjusted in the editor while building a game, or by **scripts** when running the game. There are two main types of Properties: **Values** and **References**.

Look at the image below. It is an empty GameObject with an **Audio Source** Component. All the values of the **Audio Source** in the Inspector are the default values.



This Component contains a single Reference property, and seven Value properties. **Audio Clip** is the Reference property. When this Audio Source begins playing, it will attempt to play the audio file that is referenced in the **Audio Clip** property. If no reference is made, an error will occur because there is no audio to be played. You must reference the file within the Inspector. This is as easy as dragging an audio file from the **Project View** onto the Reference Property or using the Object Selector.



Now a sound effect file is referenced in the Audio Clip property

Components can include references to any other type of Component, GameObjects, or **Assets**. You can read more about assigning references on the page about Editing Properties.

The remaining properties on the Audio Clip are all Value properties. These can be adjusted directly in the Inspector. The Value properties on the Audio Clip are all toggles, numeric values, drop-down fields, but value properties can also be text strings, colors, curves, and other types. You can read more about these and about editing value properties on the page about editing value properties.

Component Context Menu commands

The context menu for a component has a number of useful commands.

The Component Context Menu

The same commands are also available from the "gear" icon in the extreme top-right of the component's panel in the inspector.

Reset

This command restores the values the component's properties had before the most recent editing session.

Remove

A **Remove Component** command is available for cases where you no longer need the component attached to the GameObject. Note that there are some combinations of components that depend on each other (eg, Hinge Joint only works when a Rigidbody is also attached); you will see a warning message if you try to remove components that others depend on.

Move Up/Down

Use the **Move Up** and **Move Down** commands to rearrange the order of components of a GameObject in the Inspector.

Tip: Alternatively, click and drag the component's name up or down in the Inspector window, and then drop it.

Copy/Paste

The **Copy Component** command stores the type and current property settings of a component. These can then be pasted into another component of the same type with **Paste Component Values**. You can also create a new component with the copied values on an object by using **Paste Component As New**

Testing out Properties

While your game is in **Play Mode**, you are free to change properties in any GameObject's Inspector. For example, you might want to experiment with different heights of jumping. If you create a **Jump Height** property in a script, you can enter Play Mode, change the value, and press the jump button to see what happens. Then without exiting Play Mode you can change it again and see the results within seconds. When you exit Play Mode, your properties will revert to their pre-Play Mode values, so you don't lose any work. This workflow gives you incredible power to experiment, adjust, and refine your gameplay without investing a lot of time in iteration cycles. Try it out with any property in Play Mode. We think you'll be impressed.

2018–09–18 Page amended with limited editorial review

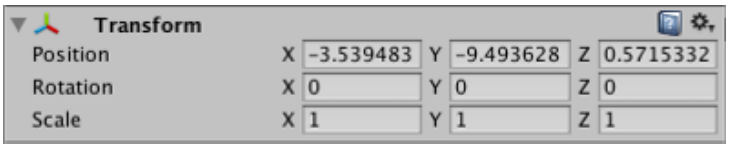
Ability to drag and drop components in Inspector window added in 5.6

Transform

Leave feedback

SWITCH TO SCRIPTING

The **Transform component** determines the **Position**, **Rotation**, and **Scale** of each object in the **scene**. Every **GameObject** has a Transform.



Properties

Property:	*Function:*
Position	Position of the Transform in X, Y, and Z coordinates.
Rotation	Rotation of the Transform around the X, Y, and Z axes, measured in degrees.
Scale	Scale of the Transform along X, Y, and Z axes. Value “1” is the original size (size at which the object was imported).

The position, rotation and scale values of a Transform are measured relative to the Transform’s parent. If the Transform has no parent, the properties are measured in world space.

Creating components with scripting

Leave feedback

Scripting (or creating scripts) is writing your own additions to the Unity Editor's functionality in code, using the Unity Scripting API.

When you create a script and attach it to a **GameObject**, the script appears in the **GameObject's Inspector** just like a built-in **component**. This is because **scripts** become **components** when you save them in your project.

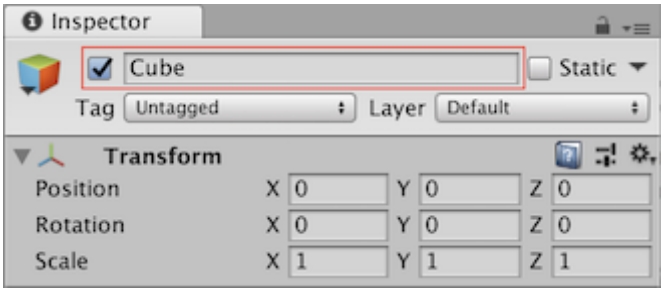
In technical terms, any script you make compiles as a type of **component**, so the Unity Editor treats your script like a built-in component. You define the members of the script to be exposed in the Inspector, and the Editor executes whatever functionality you've written.

Read more about creating and using scripts in the Scripting section of this User Manual.

Deactivating GameObjects

Leave feedback

You can mark a **GameObject** as inactive to temporarily remove it from the **Scene**. To do this, navigate to the **Inspector** and uncheck the checkbox next to the **GameObject**'s name (see image below), or use the activeSelf property in script.



A GameObject's activation checkbox next to the name, both highlighted in the red box

Deactivating a parent GameObject

When you deactivate a parent GameObject, You also deactivate all of its child GameObjects.

The deactivation overrides the `activeSelf` setting on all child GameObjects, so Unity makes the whole hierarchy inactive from the parent down. This does not change the value of the `activeSelf` property on the child GameObjects, so they return to their original state when you re-activate the parent. This means that you can't determine whether or not a child GameObject is currently active in the Scene by reading its **activeSelf** property. Instead, you should use the activeInHierarchy property, which takes the overriding effect of the parent into account.

If you want to change the **activeSelf** settings of a GameObject's child Gameobjects, but not the parent, you can use code like the following:

```
void DeactivateChildren(GameObject g, bool a)
{
    g.activeSelf = a;

    foreach (Transform child in g.transform)
    {
        DeactivateChildren(child.gameObject, a);
    }
}
```

Tags

Leave feedback

A **Tag** is a reference word which you can assign to one or more **GameObjects**. For example, you might define "Player" Tags for player-controlled characters and an "Enemy" Tag for non-player-controlled characters. You might define items the player can collect in a **Scene** with a "Collectable" Tag.

Tags help you identify GameObjects for scripting purposes. They ensure you don't need to manually add GameObjects to a script's exposed properties using drag and drop, thereby saving time when you are using the same script code in multiple GameObjects.

Tags are useful for triggers in Collider control **scripts**; they need to work out whether the player is interacting with an enemy, a prop, or a collectable, for example.

You can use the GameObject.FindWithTag() function to find a GameObject by setting it to look for any object that contains the Tag you want. The following example uses GameObject.FindWithTag(). It instantiates `respawnPrefab` at the location of GameObjects with the Tag "Respawn":

JavaScript:

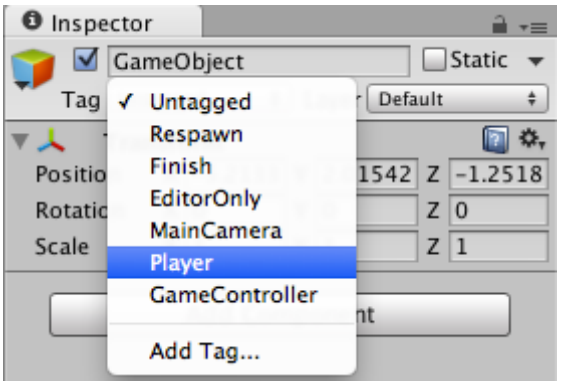
```
var respawnPrefab : GameObject;
var respawn = GameObject.FindWithTag ("Respawn");
Instantiate (respawnPrefab, respawn.position, respawn.rotation);
```

C#:

```
using UnityEngine;
using System.Collections;

public class Example : MonoBehaviour {
    public GameObject respawnPrefab;
    public GameObject respawn;
    void Start() {
        if (respawn == null)
            respawn = GameObject.FindWithTag("Respawn");

        Instantiate(respawnPrefab, respawn.transform.position, respawn.transform
    }
}
```

Creating new Tags

The **Inspector** shows the **Tag** and **Layer** drop-down menus just below any GameObject's name.



To create a new Tag, select **Add Tag...**. This opens the Tag and Layer Manager in the Inspector. Note that once you name a Tag, it cannot be renamed later.

Layers are similar to Tags, but are used to define how Unity should render GameObjects in the Scene. See documentation on the Layers page for more information.

Applying a Tag

The **Inspector** shows the **Tag** and **Layer** drop-down menus just below any GameObject's name. To apply an existing Tag to a GameObject, open the **Tags** dropdown and choose the Tag you want to apply. The GameObject is now associated with this Tag.

Hints

A GameObject can only have one Tag assigned to it.

Unity includes some built-in Tags which do not appear in the Tag Manager:

Untagged
Respawn
Finish
EditorOnly
MainCamera
Player
GameController

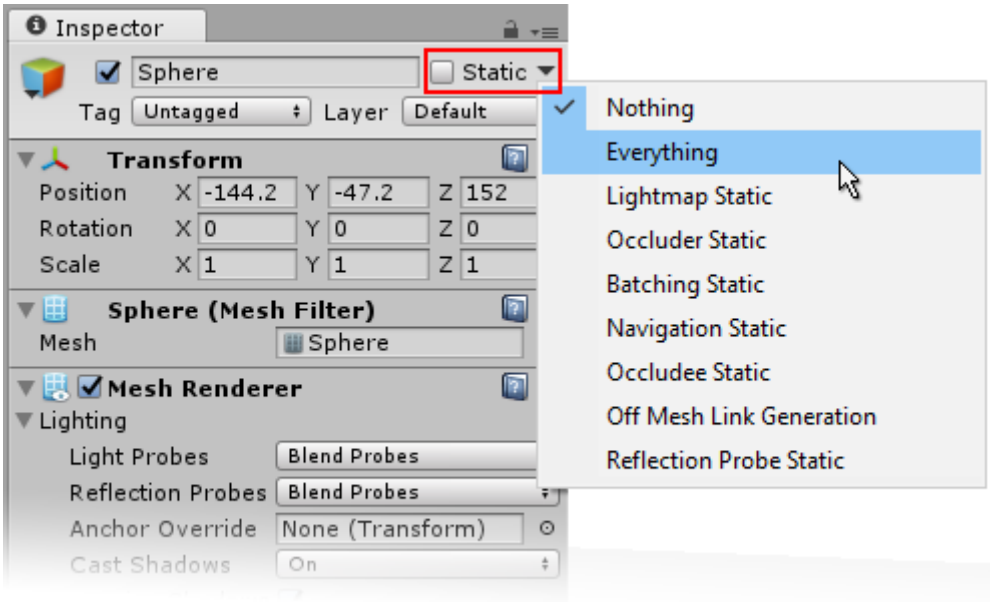
You can use any word you like as a Tag. You can even use short phrases, but you may need to widen the Inspector to see the Tag's full name.

Static GameObjects

Leave feedback

Many optimisations need to know if an object can move during gameplay. Information about a **Static** (ie, non-moving) object can often be precomputed in the editor in the knowledge that it will not be invalidated by a change in the object's position. For example, **rendering** can be optimised by combining several static objects into a single, large object known as a *batch*.

The **inspector** for a **GameObject** has a *Static* checkbox and menu in the extreme top-right, which is used to inform various different systems in Unity that the object will not move. The object can be marked as static for each of these systems individually, so you can choose not to calculate static optimisations for an object when it isn't advantageous.



The Static checkbox and drop-down menu, as seen when viewing a GameObject in the Inspector

Static Settings

The *Everything* and *Nothing* enable or disable static status simultaneously for all systems that make use of it. These systems are:

Lightmapping: advanced lighting for a **scene**;
Occluder and Occludee: rendering optimization based on the visibility of objects from specific **camera** positions;
Batching: rendering optimization that combines several objects into one larger object;
Navigation: the system that enables characters to negotiate obstacles in the scene;
Off-mesh Links: connections made by the Navigation system between discontinuous areas of the scene.
Reflection Probe: captures a spherical view of its surroundings in all directions.
See the pages about these topics for further details on how the static setting affects performance.

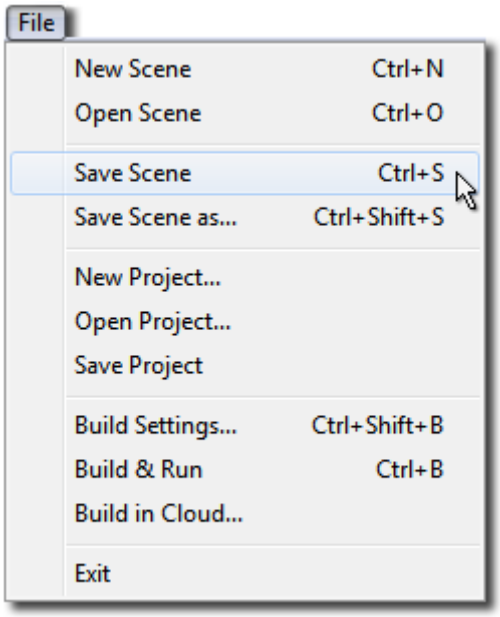
Saving Your Work

Leave feedback

Unity stores lots of different types of information about your project, and some of these are saved in different ways to others. This means that *when* your work is saved depends on what kind of changes you are making.

Of course, we recommend you save regularly, and use a Version Control System (VCS) to preserve incremental changes to your work, and allow you to try out and roll back changes without risking loss to your work.

Saving changes to the current scene ("Save Scene")

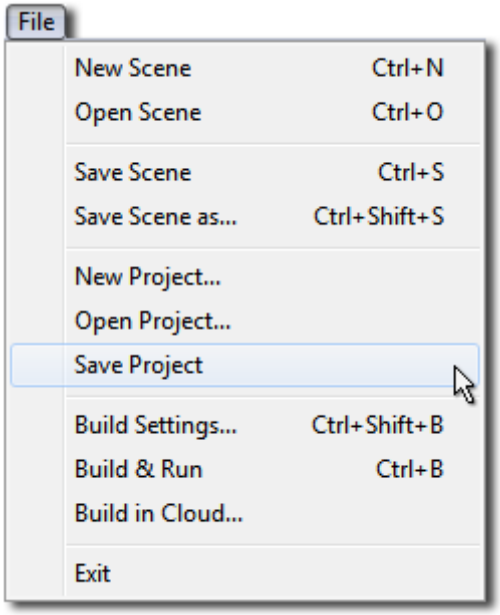


Scene changes include modifications to any objects in the Hierarchy. For example, adding, moving or deleting Game Objects, changing parameters of hierarchy Game Objects in the **inspector**.

To save changes to the **scene**, select Save **Scene** from the file menu, or hit **Ctrl/Cmd + S**. This saves current changes to the scene *and* Does a "**Save Project**" (below).

This means that when you do a "Save Scene", *everything* is saved.

Saving project-wide changes ("Save Project")



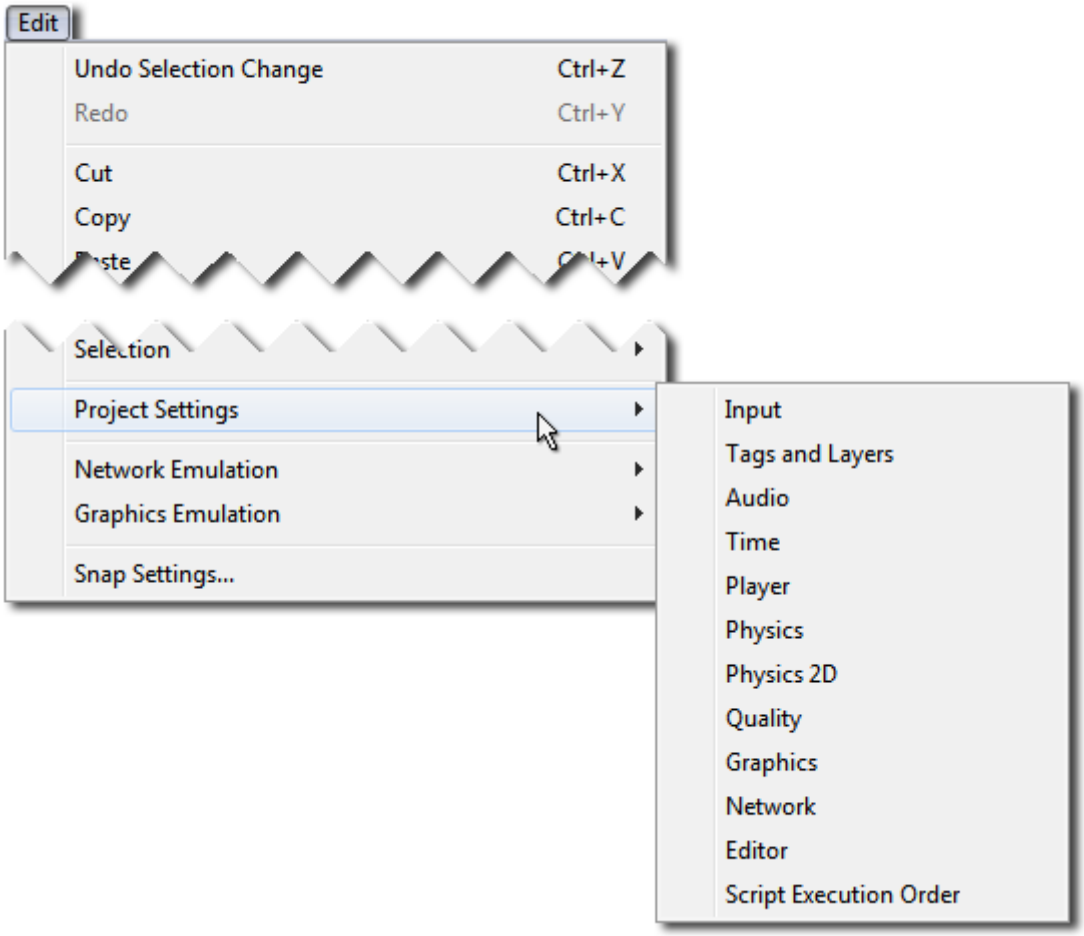
Some changes that you can make in Unity are not scene-specific, they are *project-wide*. These settings can be saved independently of the scene changes, by selecting "Save Project" from the file menu.

Using "Save Project" does not save changes to your Scene, only the project-wide changes are saved. You may want to, for instance, save your project but not changes to your scene if you have used a temporary scene to make some changes to a **prefab**.

The project-wide changes which are saved when you "Save Project" include:

All the "Project Settings":

All the settings for each of the "**Project Settings**" menu items, such as custom input axes, user-defined tags or layers, and the physics gravity strength are saved when you "Save Project".



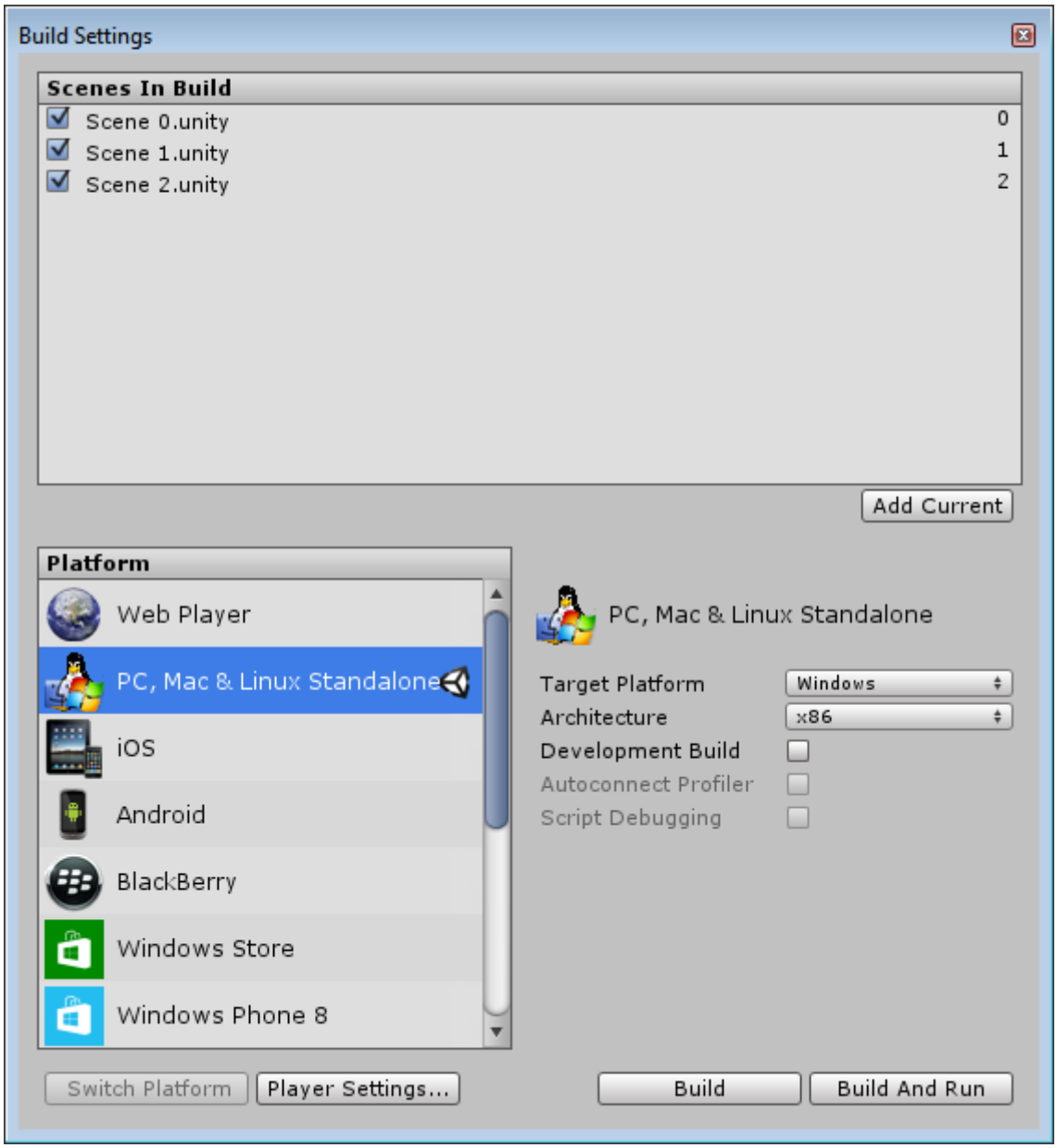
The Project Settings menu

Changes to these settings are saved in the Library folder when the Project is saved:

Input: saved as ´InputManager.**asset**´
Tags And Layers: saved as ´TagManager.asset´
Audio: saved as ´AudioManager.asset´
Time: saved as ´TimeManager.asset´
Player: saved as ´ProjectSettings.asset´
Physics: saved as ´DynamicsManager.asset´
Physics 2D: saved as ´Physics2DSettings.asset´
Quality: saved as ´QualitySettings.asset´
Graphics: saved as ´GraphicsSettings.asset´
Network: saved as ´NetworkManager.asset´
Editor: saved as ´EditorUserSettings.asset´

The “Build Settings”

Build Settings are also saved in the Library folder as ´EditorBuildSettings.asset´.



The Build Settings are saved when you “Save Project”

Changes to assets in the Project Window

Also saved along with project-wide settings are changes to assets *that do not have an “apply” button*, for example changes to any of the following:

Material parameters
Prefabs
Animator Controllers (state machines)
Avatar Masks
Any other asset changes that do not have an “apply” button

Changes that are immediately written to disk (no save required):

There are some types of change which are immediately written to disk without the need to perform a “Save” action at all. These include the following:

Changes to any import setting requiring the user to press an “apply” button

The import settings for most asset types require that you press an “Apply” button for the changes to take effect. This causes the asset to be re-imported according to the new settings. These changes are saved immediately when you hit the Apply button. For example:

Changing the texture type of an image asset
Changing the scale factor of an 3D model asset
Changing the **compression** settings of an audio asset
Any other import setting change which has an “apply” button

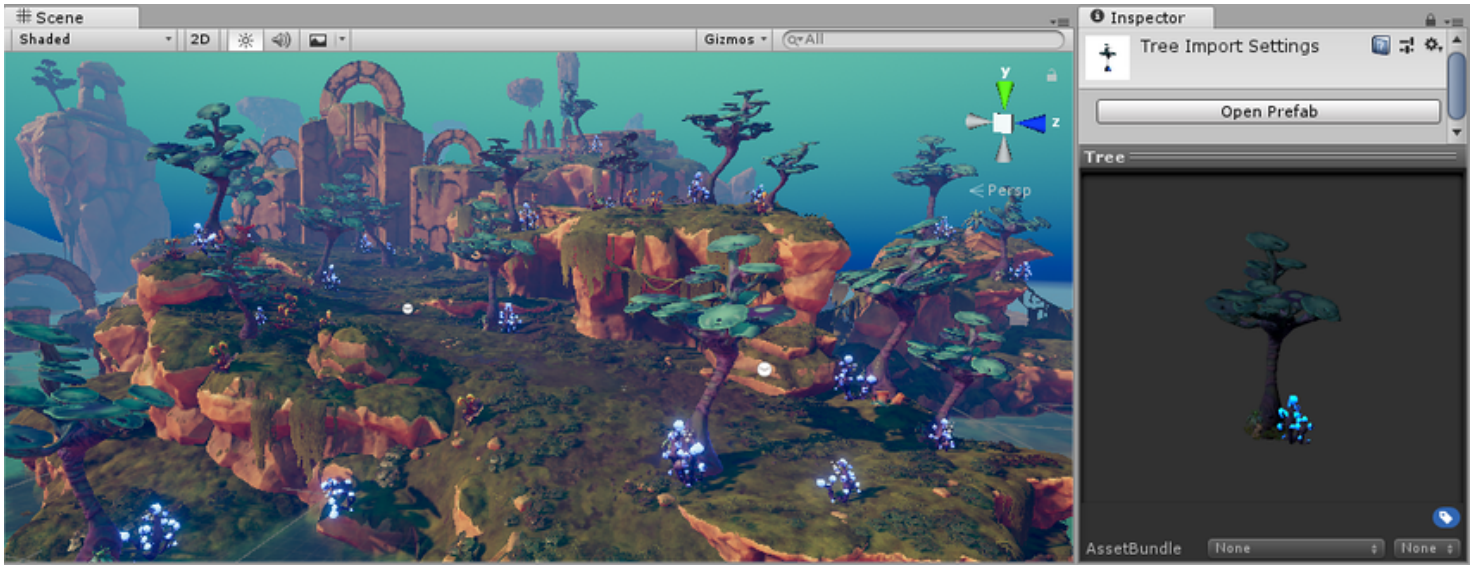
Other changes which are saved immediately:

A few other types of data are saved to disk immediately or automatically without the need to perform a “Save” action:

The creation of new assets, eg: new materials or prefabs (But not subsequent changes to those assets)
Baked Lighting data (saved when the bake completes).
Baked navigation data (saved when the bake completes)
Baked **occlusion culling** data (saved when the bake completes)
Script execution order changes (after “apply” is pressed, this data is saved in each script’s .meta file)

Prefabs

Leave feedback



Unity's **Prefab** system allows you to create, configure, and store a GameObject complete with all its **components**, property values, and child **GameObjects** as a reusable **Asset**. The **Prefab Asset** acts as a template from which you can create new **Prefab** instances in the **Scene**.

When you want to reuse a GameObject configured in a particular way – like a non-player character (NPC), prop or piece of scenery – in multiple places in your Scene, or across multiple Scenes in your Project, you should convert it to a Prefab. This is better than simply copying and pasting the GameObject, because the Prefab system allows you to automatically keep all the copies in sync.

Any edits that you make to a Prefab Asset are automatically reflected in the instances of that Prefab, allowing you to easily make broad changes across your whole Project without having to repeatedly make the same edit to every copy of the Asset.

You can nest Prefabs inside other Prefabs to create complex hierachies of objects that are easy to edit at multiple levels.

However, this does not mean all Prefab instances have to be identical. You can override settings on individual prefab instances if you want some instances of a Prefab to differ from others. You can also create variants of Prefabs which allow you to group a set of overrides together into a meaningful variation of a Prefab.

You should also use Prefabs when you want to instantiate GameObjects at runtime that did not exist in your Scene at the start - for example, to make powerups, special effects, projectiles, or NPCs appear at the right moments during gameplay.

Some common examples of Prefab use include:

Environmental Assets - for example a certain type of tree used multiple times around a level (as seen in the screenshot above).

Non-player characters (NPCs) - for example a certain type of robot may appear in your game multiple times, across multiple levels. They may differ (using *overrides*) in the speed they move, or the sound they make.

Projectiles - for example a pirate's cannon might instantiate a cannonball Prefab each time it is fired.

The player's main character - the player prefab might be placed at the starting point on each level (separate Scenes) of your game.

2018–07–31 Page amended with limited editorial review

Updated to include improved prefab features - Nested Prefabs and Prefab Variants added in 2018.3

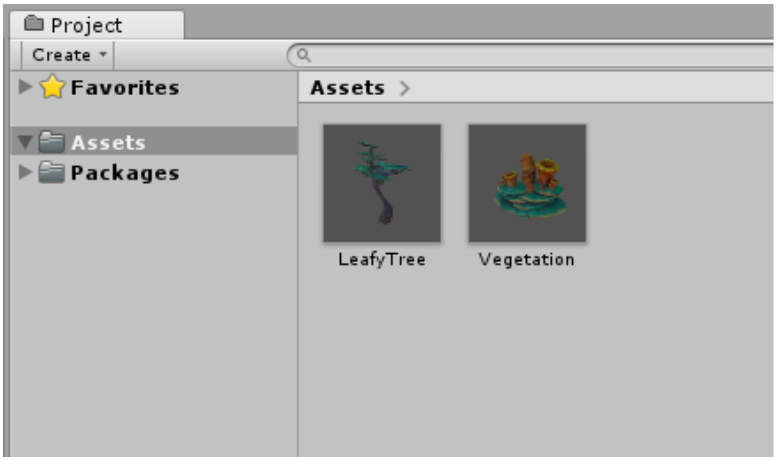
Creating Prefabs

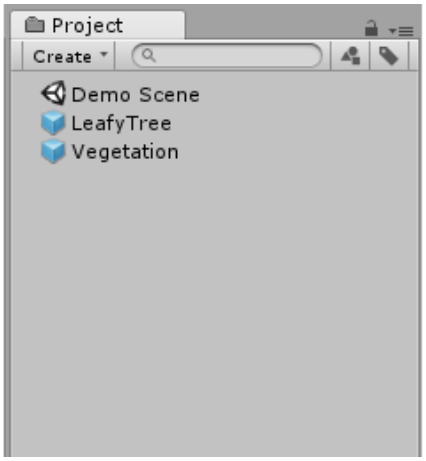
Leave feedback

In Unity's **Prefab** system, **Prefab Assets** act as templates. You create **Prefab** Assets in the Editor, and they are saved as an **Asset** in the Project window. From **Prefab Assets**, you can create any number of **Prefab instances**. **Prefab** instances can either be created in the editor and saved as part of your **Scenes**, or instantiated at runtime.

Creating Prefab Assets

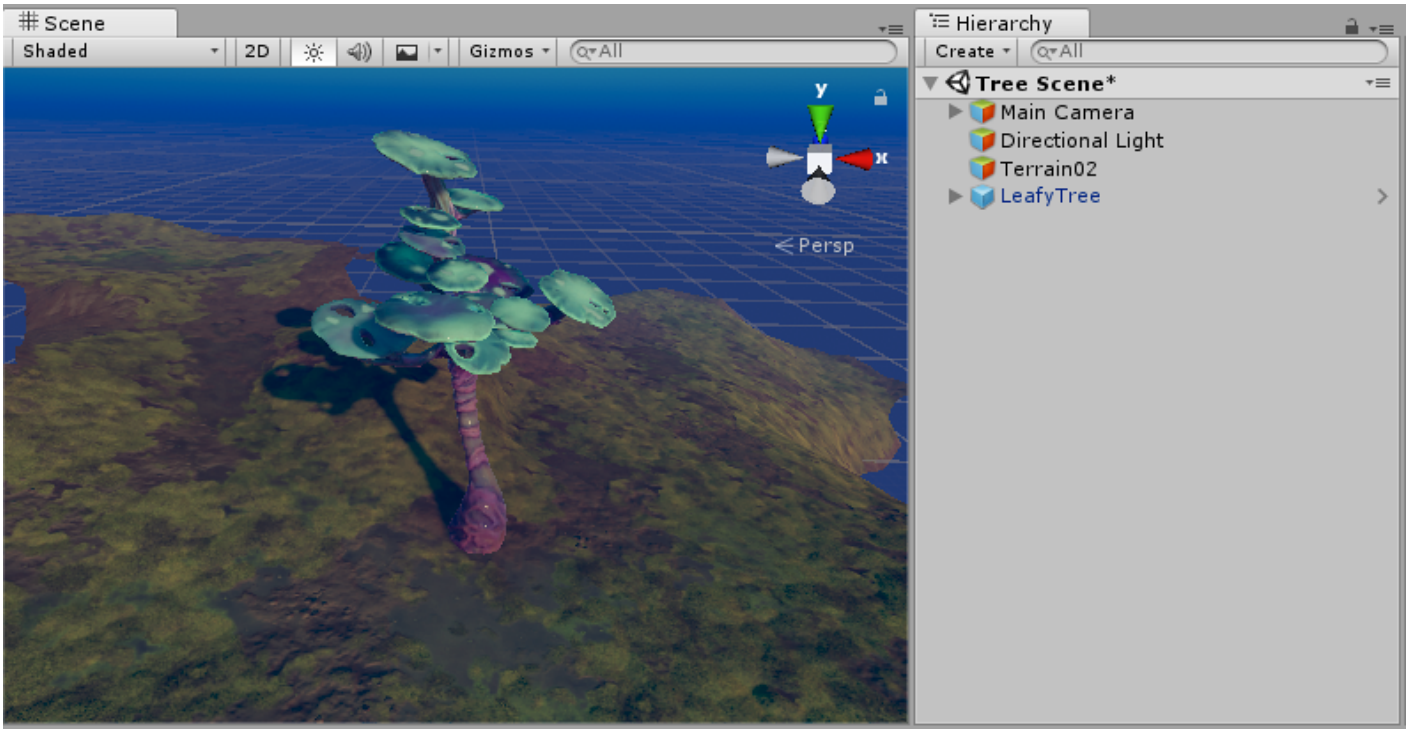
To create a Prefab Asset, drag a GameObject from the Hierarchy window into the Project window. The GameObject, and all its **components** and child **GameObjects**, becomes a new Asset in your Project window. Prefabs Assets in the Project window are shown with a thumbnail view of the GameObject, or the blue cube Prefab icon, depending on how you have set up your Project window.





Two prefabs ("LeafyTree" and "Vegetation") shown in the Project window in two-column view (left) and one-column view (right)

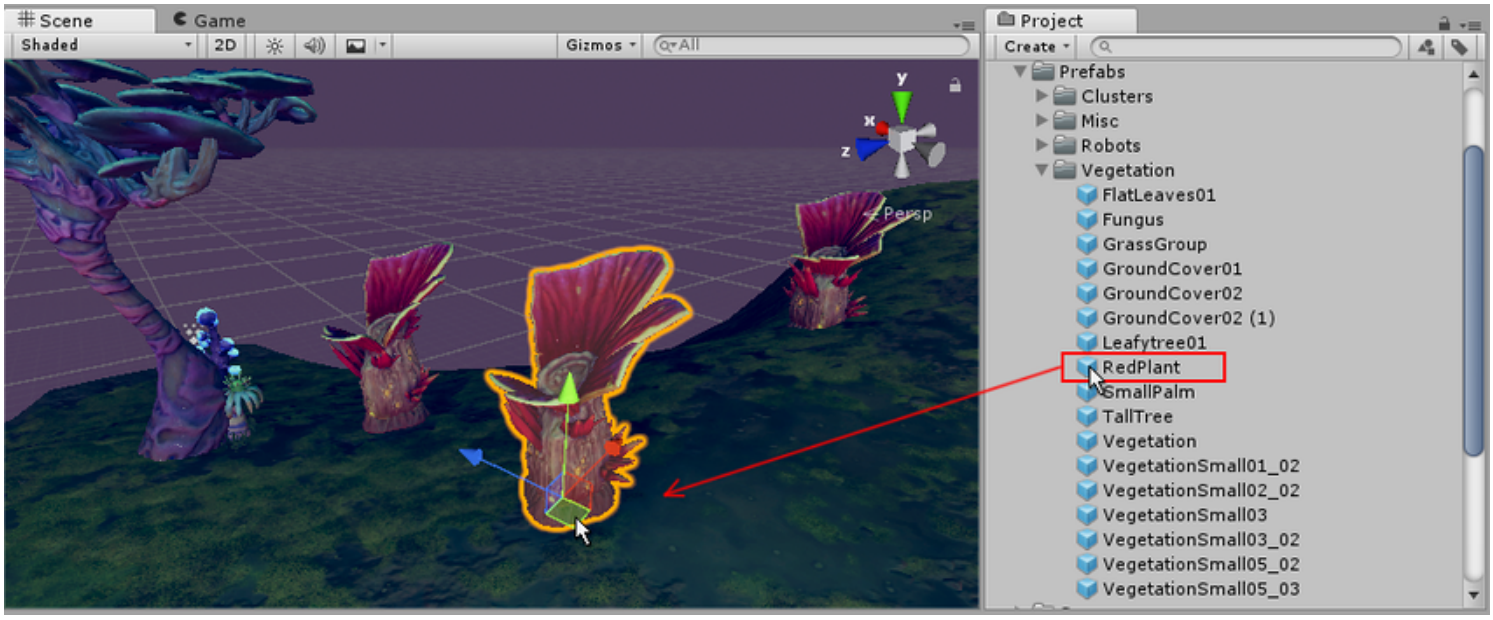
This process of creating the Prefab Asset also turns the original GameObject into a Prefab instance. It is now an instance of the newly created Prefab Asset. Prefab instances are shown in the Hierarchy in blue text, and the root GameObject of the Prefab is shown with the blue cube Prefab icon, instead of the red, green and blue GameObject icon.



A Prefab instance (LeafyTree) in the scene

Creating Prefab instances

You can create instances of the Prefab Asset in the Editor by dragging the Prefab Asset from the **Project view** to the Hierarchy or **Scene view**.



Dragging a Prefab “RedPlant” into the Scene

You can also create instances of Prefabs at runtime using scripting. For more information, see Instantiating Prefabs.

2018–07–31 Page published with limited editorial review

Nested Prefabs and Prefab Variants added in 2018.3

Editing a Prefab in Prefab Mode

Leave feedback

To edit a **Prefab Asset**, open it in **Prefab** Mode. **Prefab** Mode allows you to view and edit the contents of the **Prefab Asset** in isolation, separately from any other objects in your **Scene**. Changes you make in **Prefab** Mode affect all instances of that **Prefab**.

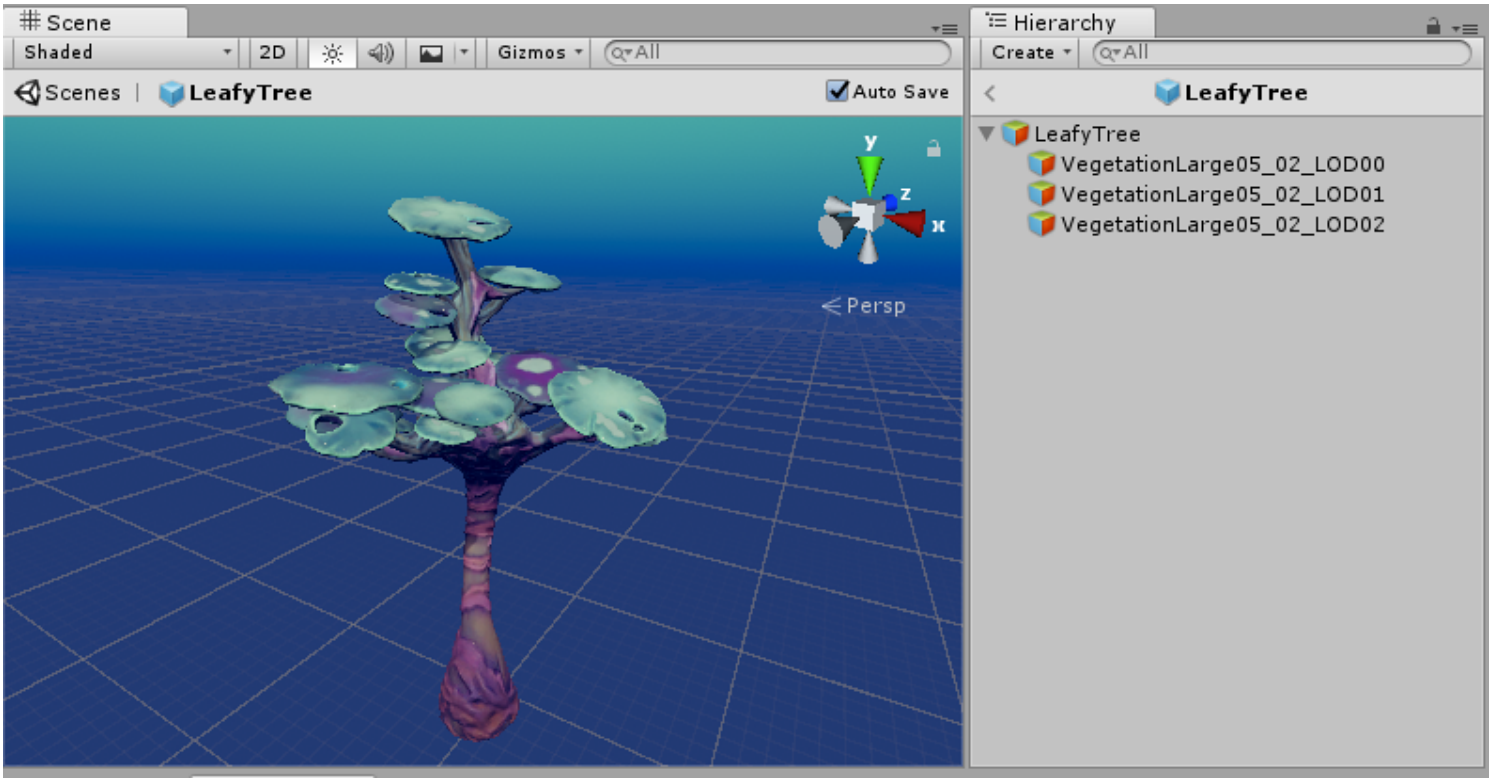
Entering and exiting Prefab Mode

There are many ways to begin editing a Prefab in Prefab Mode, including:

Double-clicking it in the Project window
Using the arrow button next to the Prefab in the Hierarchy window
Clicking the “Open” button in the **Inspector** window of a Prefab Asset

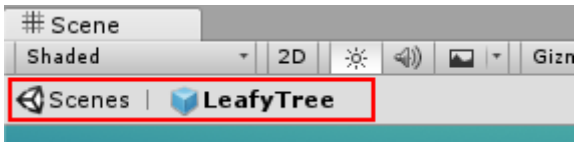
The arrow button next to a Prefab in the Hierarchy window

Entering Prefab Mode makes the **Scene View** and the Hierarchy window show only the contents of that Prefab. Here, the root of the Prefab is a regular **GameObject** - it doesn’t have the blue Prefab instances icon.



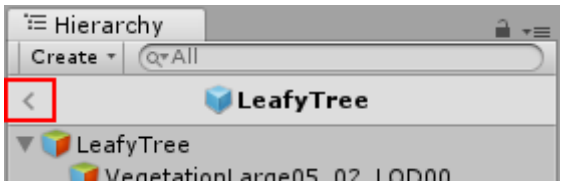
The Scene View and Hierarchy in Prefab Mode

In Prefab Mode, the Scene View displays a breadcrumb bar at the top. The rightmost entry is the currently open Prefab. Use the breadcrumb bar to navigate back to the main Scenes or other Prefab Assets that you might have opened.



The breadcrumb bar at the top of the Scene view, visible when in Prefab Mode

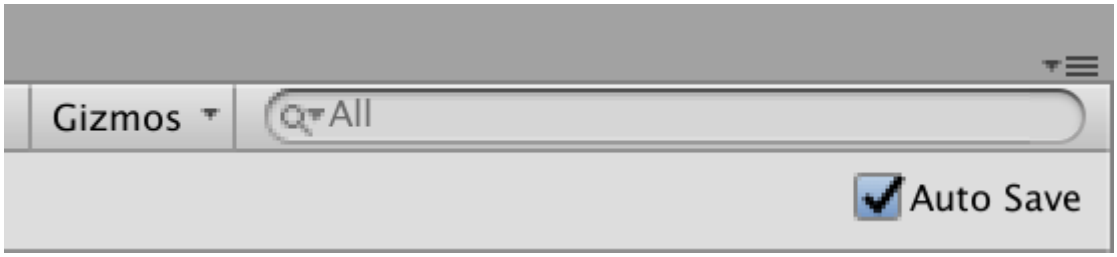
Additionally, the Hierarchy window displays a Prefab header bar at the top which shows the currently open Prefab. You can use the back arrow in the header bar to navigate back one step, which is equivalent to clicking the previous breadcrumb in the breadcrumb bar in the Scene View.



The back arrow in the header bar of the Hierarchy window, visible when in Prefab Mode

Auto Save

Prefab Mode has an **Auto Save** setting in the top right corner of the Scene View. When it is enabled, any changes that you make to a Prefab are automatically saved to the Prefab Asset. *Auto Save* is on by default.



The Auto Save toggle in the upper right corner of the Scene View in Prefab Mode

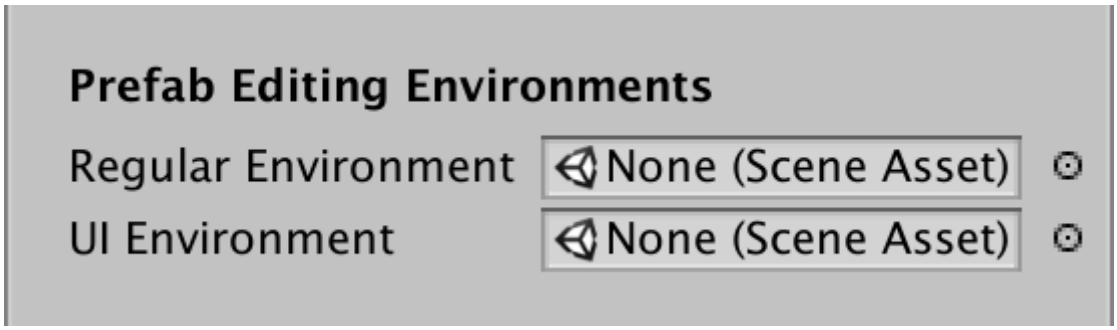
If you want to make changes without automatically saving those changes to the Preset Asset, turn **Auto Save** off. In this case, you are asked if you want to save unsaved changes or not when you go out of Prefab Mode for the current Prefab. If editing a Prefab in Prefab Mode seems slow, turning off **Auto Save** may help.

Editing Environment

You can assign a Scene as an **editing environment** to Prefab Mode, which allows you to edit your Prefab against a backdrop of your choosing rather than an empty Scene. This can be useful for seeing how your Prefab looks against typical scenery in your game.

The objects in the Scene that you assign as the editing environment are not selectable when in Prefab Mode, nor will they show in the Hierarchy. This is to allow you to focus on editing your Prefab without accidently selecting other unrelated objects, and without having a cluttered Hierarchy window.

To set a Scene as the editing environment, open the Editor settings (**Edit > Project Settings > Editor**) and go to the **Prefab Editing Environment** section. Use the **Regular Environment** setting for “non-UI” Prefabs, and the **UI Environment** setting for UI Prefabs. UI prefabs are those Prefabs that have a Rect Transform component on the root, rather than a regular Transform component. “non-UI” Prefabs are those which have a regular Transform component.



Prefab editing environment settings in the Editor Project Settings

---

2018–07–31 Page published with limited editorial review

Nested Prefabs and Prefab Variants added in 2018.3

Instance overrides

Leave feedback

Instance overrides allow you to create variations between **Prefab** instances, while still linking those instances to the same **Prefab Asset**.

When you modify a Prefab Asset, the changes are reflected in all of its instances. However, you can also make modifications directly to an individual instance. Doing this creates an **instance override** on that particular instance.

An example would be if you had a Prefab Asset “Robot”, which you placed in multiple levels in your game. However, each instance of the “Robot” has a different speed value, and a different **audio clip** assigned.

There are four different types of **instance override**:

Overriding the value of a property

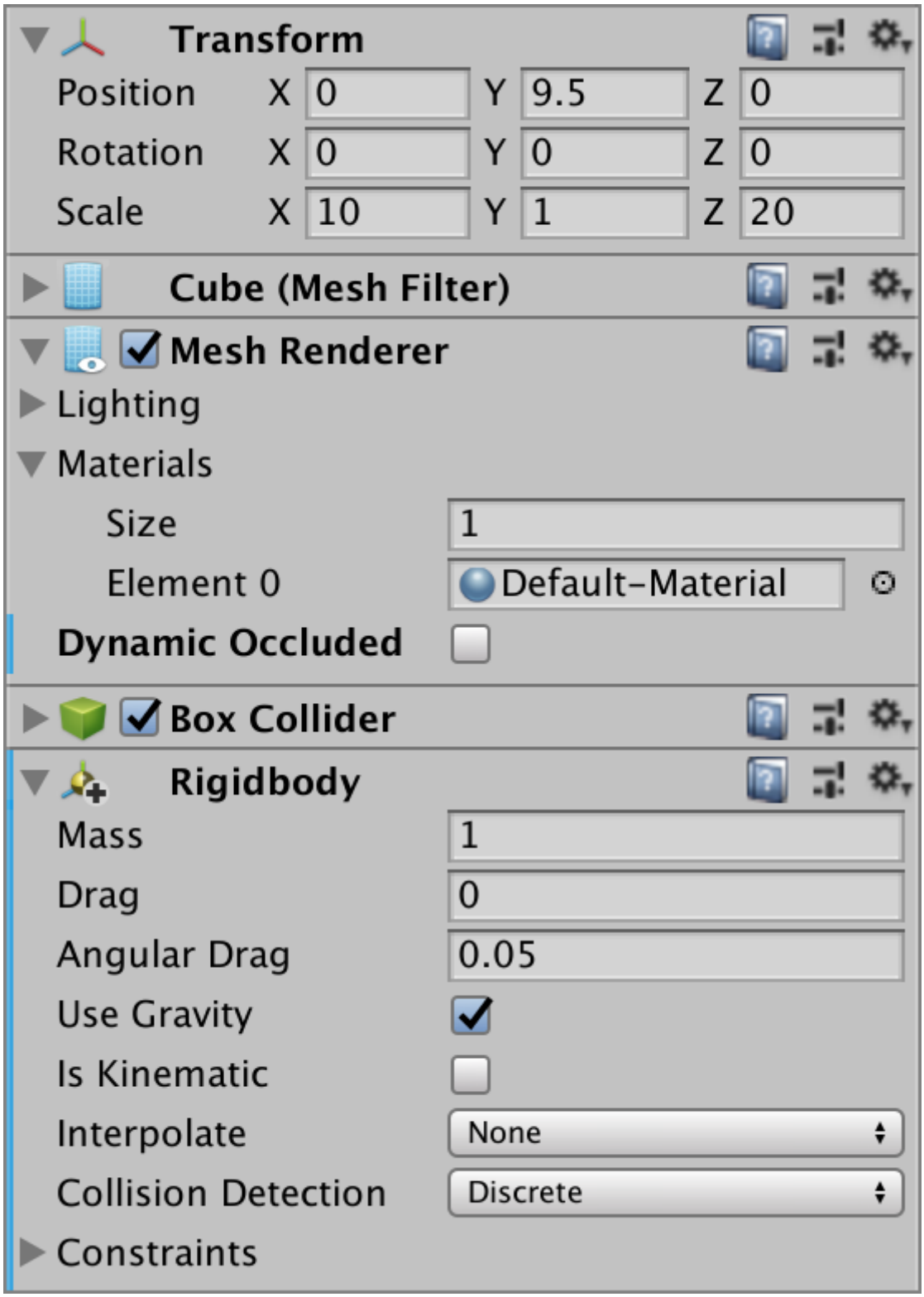
Adding a component

Removing a component

Adding a child GameObject

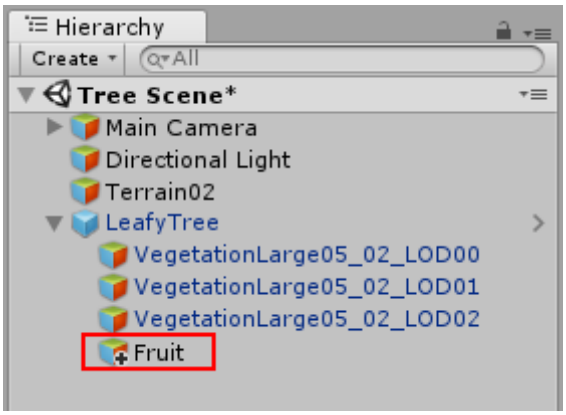
There are some limitations with Prefab instances: you cannot reparent a **GameObject** that is part of a Prefab, and you cannot remove a **GameObject** that is part of the Prefab. You can, however, deactivate a **GameObject**, which is a good substitute for removing a **GameObject** (this counts as a property override).

In the **Inspector** window, instance overrides are shown with their name label in **bold**, and with a blue line in the left margin. When you add a new **component** a Prefab instance, the blue line in the margin spans the entire **component**.



The inspector windows showing a Prefab instance with overridden “Dynamic Occluded” property and a Rigidbody component added as an override.

Added and removed components also have plus and minus badges on their icons in the Inspector, and added GameObjects have a plus badge on their icon in the Hierarchy.



The Hierarchy window showing a Prefab instance with a child GameObject called “Fruit” added as an override.

Overrides take precedence

An overridden property value on a Prefab instance always takes precedence over the value from the Prefab Asset. This means that if you change a property on a Prefab Asset, it doesn’t have any effect on instances where that property is overridden.

If you make a change to a Prefab Asset, and it does not update all instances as expected, you should check whether that property is overridden on the instance. It is best to only use instance overrides when strictly necessary, because if you have a large number of instance overrides throughout your Project, it can be difficult to tell whether your changes to the Prefab Asset will or won’t have an effect on all of the instances.

Alignment is specific to Prefab instance

The **alignment** of a Prefab instance is a special case, and is handled differently to other properties. The **alignment** values are never carried across from the Prefab Asset to the Prefab instances. This means they can always differ from the Prefab Asset’s alignment without it being an explicit instance override. Specifically, the alignment means the **Position** and **Rotation** properties on the **root Transform** of the Prefab instance, and for a Rect Transform this also includes the **Width**, **Height**, **Margins**, **Anchors** and **Pivot** properties.

This is because it is extremely rare to require multiple instances of a Prefab to take on the same position and rotation. More commonly, you will want your prefab instances to be at different positions and rotations, so Unity does not count these as Prefab overrides.

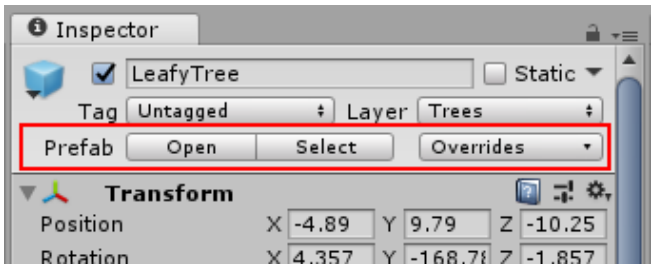
2018–07–31 Page published with limited editorial review

Nested Prefabs and Prefab Variants added in 2018.3

Editing a Prefab via its instances

Leave feedback

The **Inspector** for the root of a **Prefab** instance has three more controls than a normal **GameObject**: **Open**, **Select** and **Overrides**.



The three Prefab controls in the Inspector window for a Prefab instance

The **Open** button opens the Prefab **Asset** that the instance is from in Prefab Mode, allowing you to edit the Prefab **Asset** and thereby change all of its instances. The **Select** button selects the Prefab **Asset** that this instance is from in the Project window. The **Overrides** button opens the overrides drop-down window.

Overrides dropdown

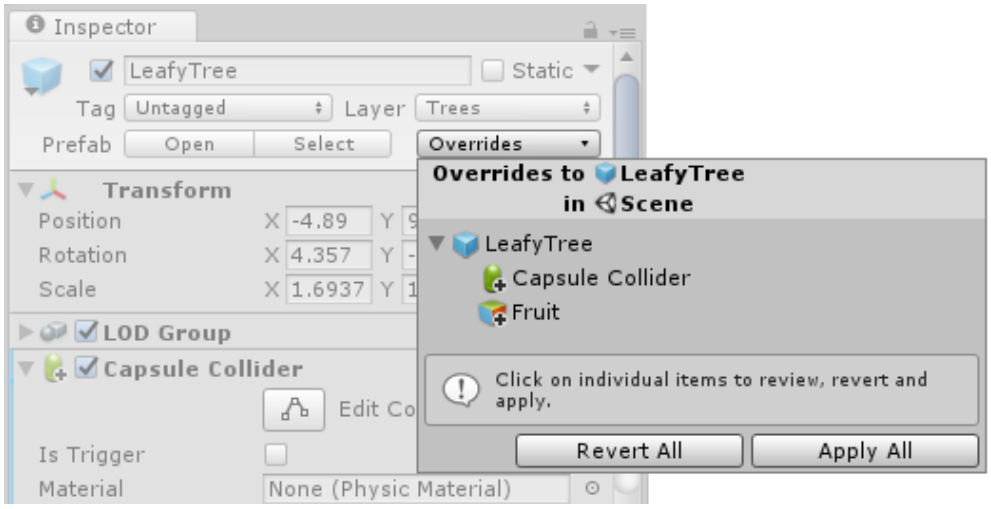
The **Overrides** drop-down window shows all the overrides on the Prefab instance. It also lets you apply overrides from the instance to the Prefab Asset, or revert overrides on the instance back to the values on the Prefab Asset. The **Overrides** drop-down button only appears for the root Prefab instance, not for Prefabs that are inside other Prefabs.

The **Overrides** drop-down window allows you to apply or revert individual prefab overrides, or apply or revert all the prefab overrides in one go.

Applying an override modifies the Prefab Asset. This puts the override (which is currently only on your Prefab instance) onto the Asset. This means the Prefab Asset now has that modification, and the Prefab instance no longer has that modification as an override.

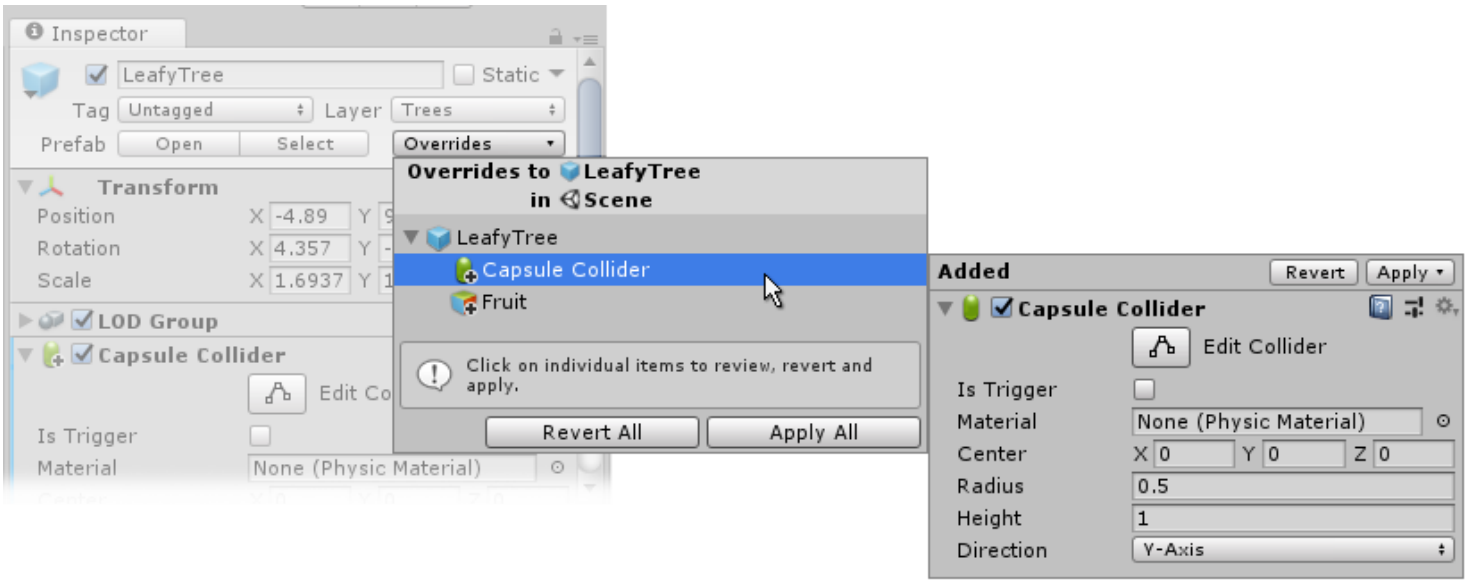
Reverting an override modifies the Prefab instance. This essentially discards your override and reverts it back to the state of the Prefab Asset.

The drop-down window shows a list of changes on the instance in the form of modified, added and removed **components**, and added GameObjects (including other Prefabs).



The Overrides dropdown in the Inspector window when viewing a Prefab instance

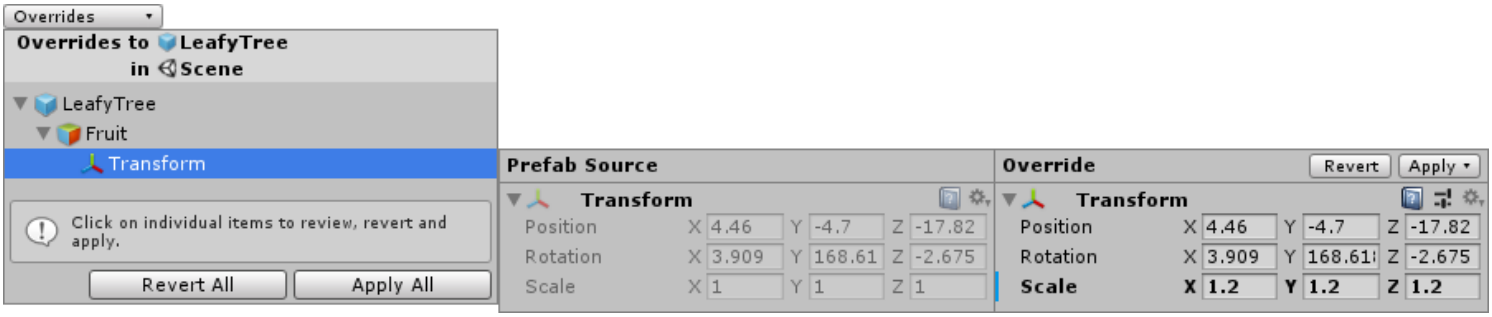
To inspect an entry, click it. This displays a floating view that shows the change and allows you to revert or apply that change.



The overrides dropdown window with an added component override selected

For components with modified values, the view displays a side-by-side comparison of the component's values on the Prefab Asset and the modified component on the Prefab instance. This allows you to compare the original Prefab Asset values with the current overrides, so that you can decide whether you would like to revert or apply those values.

In the example below, the "Fruit" child GameObject exists on both the Prefab Asset and the Prefab instance, however its scale has been increased on the instance. This increase in scale is an instance override, and can be seen as a side-by-side comparison in the **Overrides** drop-down window.



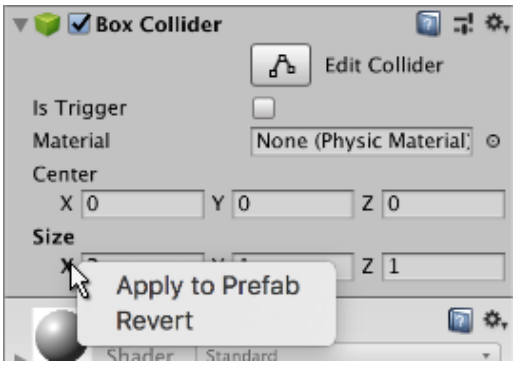
The Overrides dropdown with comparison view, showing modified values in the Transform component of a child GameObject of the prefab instance

The **Overrides** drop-down window also has **Revert All** and *Apply All*_ buttons for reverting or applying all changes at once. If you have Prefabs inside other Prefabs, the **Apply All** button always applies to the outermost Prefab, which is the one that has the **Overrides** drop-down button on its root GameObject.

Context menus

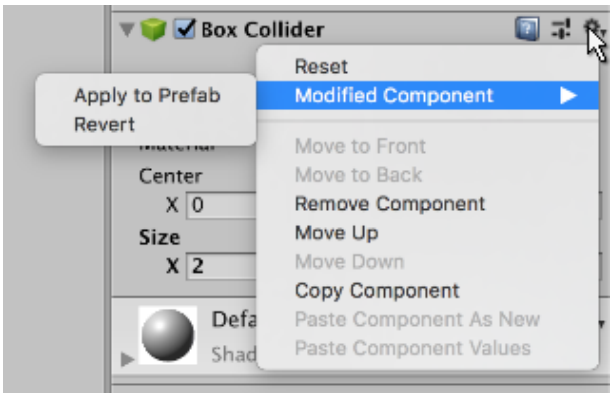
You can also **revert** and **apply** individual overrides using the context menu in the Inspector, instead of using the Overrides drop-down window.

Overridden properties are shown in bold. They can be reverted or applied via a context menu:



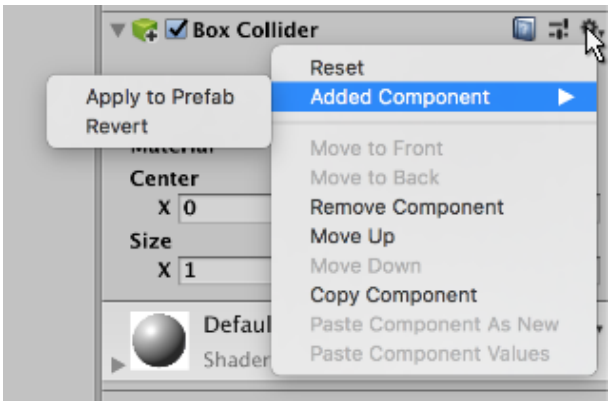
Context menu for a single property

Modified components can be reverted or applied via the cog drop-down button or context menu of the component header:



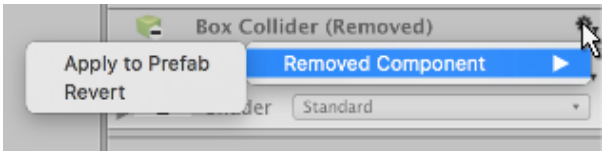
Context menu for modified component

Added components have a plus badge that overlays the icon. They can be reverted or applied via the cog drop-down button or context menu of the component header:



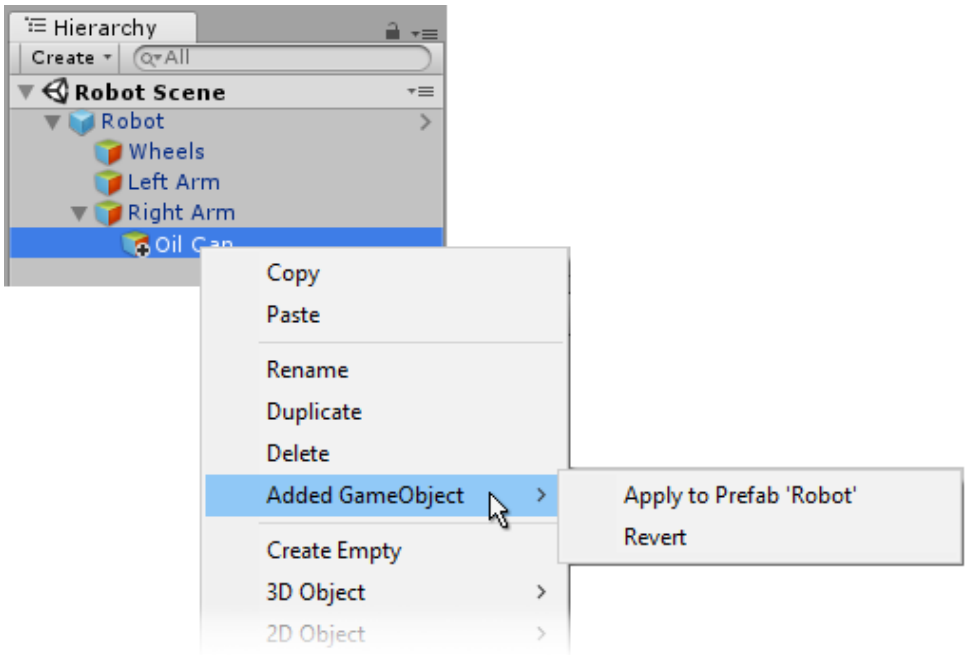
Context menu for added component

Removed components have a minus badge that overlays the icon. The removal can be reverted or applied via the cog drop-down button or context menu of the component header. Reverting the removal puts the component back, and applying the removal deletes it from the Prefab Asset as well:



Context menu for removed component

GameObjects (including other Prefabs) that are added as children to a Prefab instance have a plus badge that overlays the icon in the Hierarchy. They can be reverted or applied via the context menu on the object in the Hierarchy:



Context menu for added GameObject child

2018–07–31 Page published with limited editorial review

Nested Prefabs and Prefab Variants added in 2018.3

Nested Prefabs

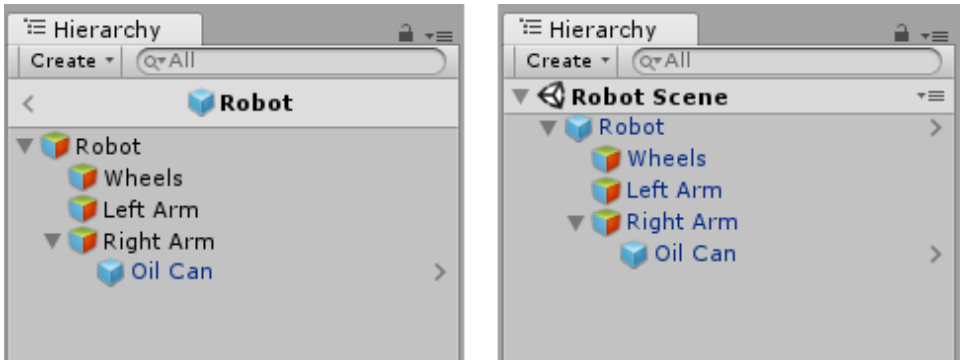
Leave feedback

You can include **Prefab** instances inside other Prefabs. This is called **nesting** Prefabs. Nested Prefabs retain their links to their own **Prefab** Assets, while also forming part of another **Prefab Asset**.

Adding a nested Prefab in Prefab Mode

In Prefab Mode, you can add and work with Prefab instances just like you would do in **Scenes**. You can drag a Prefab Asset from the Project window to the Hierarchy window or **Scene view** to create a Prefab instance from that Asset inside the Prefab you have open.

Note: The root **GameObject** of the Prefab that is open in Prefab Mode is not shown with the blue cube Prefab icon, however any instances of other Prefabs are. You can also add overrides to these Prefab instances, just like with Prefab instances in scenes.

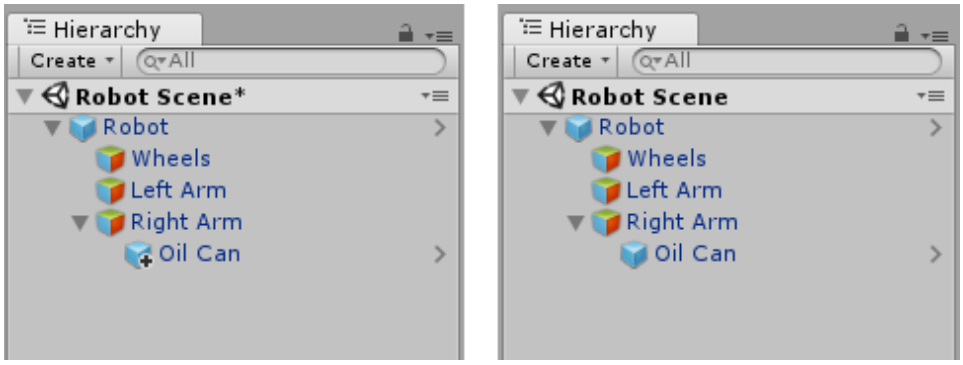


Left: “Oil Can” Prefab included (nested) in the “Robot” Prefab in Prefab Mode. Right: The “Robot” Prefab instance in the Scene with the “Oil Can” included.

Nesting Prefabs via their instances

You can also add a Prefab instance as a child to another Prefab instance in the Scene without going into Prefab Mode, just like you can add any other GameObject. Such an added Prefab instance has a plus badge overlayed on the icon in the Hierarchy which indicates that it’s an override on that specific instance of the outer Prefab.

The added Prefab can be reverted or applied to the outer Prefab in the same way as other overrides (either via the Overrides drop-down window, or via the context menu on the GameObject in the Hierarchy), as described in Editing a Prefab via its instances. The Overrides drop-down button is only on the outer Prefab. Once applied, the Prefab no longer shows the plus badge, since it is no longer an override, but is nested in the outer Prefab Asset itself. It does however retain its blue cube icon because it is a Prefab instance in its own right, and retains its connection to its own Prefab Asset.



Left: An “Oil Can” Prefab added to an instance of the “Robot” Prefab as an override. Right: The “Oil Can” Prefab has been applied to “Robot” Prefab, and is now a nested Prefab in the “Robot” Prefab Asset.

2018–07–31 Page published with limited editorial review

Nested Prefabs and Prefab Variants added in 2018.3

Prefab Variants

Leave feedback

Prefab Variants are useful when you want to have a set of predefined variations of a **Prefab**.

For example, you might want to have several different types of robot in your game, which are all based on the same basic robot Prefab. However you may want some robots to carry items, some to move at different speeds, or some to emit extra sound effects.

To do this, you could set up your initial robot Prefab to perform all the basic actions you want all robots to share, then you could create several Prefab Variants to:

Make a robot move faster by using a property override on a script to change its speed.

Make a robot carry an item by attaching an additional **GameObject** to its arm.

Give robots a rusty **joint** by adding an AudioSource **component** that plays a rusty squeaking sound.

A Prefab Variant inherits the properties of another Prefab, called the base. Overrides made to the Prefab Variant take precedent over the base Prefab's values. A Prefab Variant can have any other Prefab as its base, including Model Prefabs or other Prefab Variants.

Creating a Prefab Variant

There are multiple ways to create a Prefab Variant based on another Prefab.

You can **right-click** on a Prefab in the Project view and select **Create > Prefab Variant**. This creates a variant of the selected Prefab, which initially doesn't have any overrides. You can open the Prefab Variant in Prefab Mode to begin adding overrides to it.

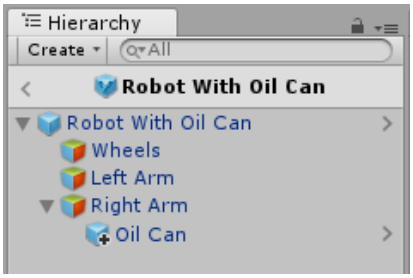
You can also **drag** a Prefab instance in the Hierarchy into the Project window. When you do this, a dialog asks if you want to create a new Original Prefab or a Prefab Variant. If you choose Prefab Variant you get a new Prefab Variant based on the Prefab instance you dragged. Any overrides you had on that instance are now inside the new Prefab Variant. You can open it in Prefab Mode to add additional overrides or edit or remove overrides.

Prefab Variants are shown with the blue Prefab icon decorated with arrows.

A basic Robot Prefab, and a variant of that Prefab called "Robot With Oil Can", as viewed in the Hierarchy window.

Editing a Prefab Variant

When a Prefab Variant is opened in Prefab Mode, the root appears as a Prefab instance with the blue Prefab icon. This Prefab instance represents the base Prefab that the Prefab Variant inherits from; it doesn't represent the Prefab Variant itself. Any edits you make to the Prefab Variant become overrides to this base that exists in the Variant.



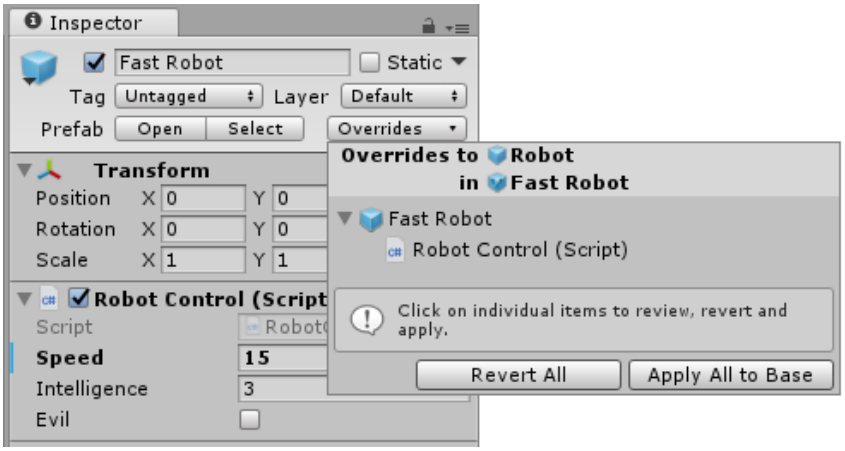
The Prefab Variant "Robot With Oil Can" n Prefab Mode. The "Oil Can" Prefab is added as an override to the base Prefab

In the screenshot above, if you were to select the **Robot With Oil Can** root GameObject and click the **Select** button in the **Inspector**, it will select the base Prefab **Robot** and not the Variant **Robot With Oil Can** because the Prefab instance is an instance of the base Prefab **Robot** and the **Select** button always selects the Prefab **Asset** that an instance comes from.

As with any Prefab instance, you can use prefab overrides in a Prefab Variant, such as modified property values, added components, removed components, and added child GameObjects. There are also the same limitations: you cannot reparent GameObjects in the Prefab Variant which come from its base Prefab. You also cannot remove a GameObject from a Prefab Variant which exists in its base Prefab. You can, however, deactivate GameObjects (as a property override) to achieve the same effect as removing a GameObject.

Note: When editing a Prefab Variant in Prefab Mode, you should understand that applying these overrides (via the Overrides drop-down window or context menus) will cause your variant's variations to be applied to the base Prefab Asset. This is often **not** what you want. The point of a Prefab Variant is to provide a convenient way to store a meaningful and reusable collection of overrides, which is why they should normally remain as overrides and not get applied to the base Prefab Asset. To illustrate this point, if you were to apply the additional **Oil Can** GameObject to the base Prefab Asset (the "Basic Robot"), then the Prefab Asset would also have the **Oil Can**. The whole point of the **Robot With Oil Can** variant is that only this variation carries an oil can, so the added **Oil Can** GameObject should be left as an override inside the Prefab Variant.

When you open the Overrides drop-down window, you can always see in its header which object the overrides are to, and in which context the overrides exist. For a Prefab Variant, the header will say that the overrides are to the base Prefab and exist in the Prefab Variant. To make it extra clear, the **Apply All** button also says **Apply All to Base**.



Overrides dropdown for a Prefab Variant when editing the Prefab Variant in Prefab Mode

---

2018–07–31 Page published with limited editorial review

Nested Prefabs and Prefab Variants added in 2018.3

Overrides at multiple levels

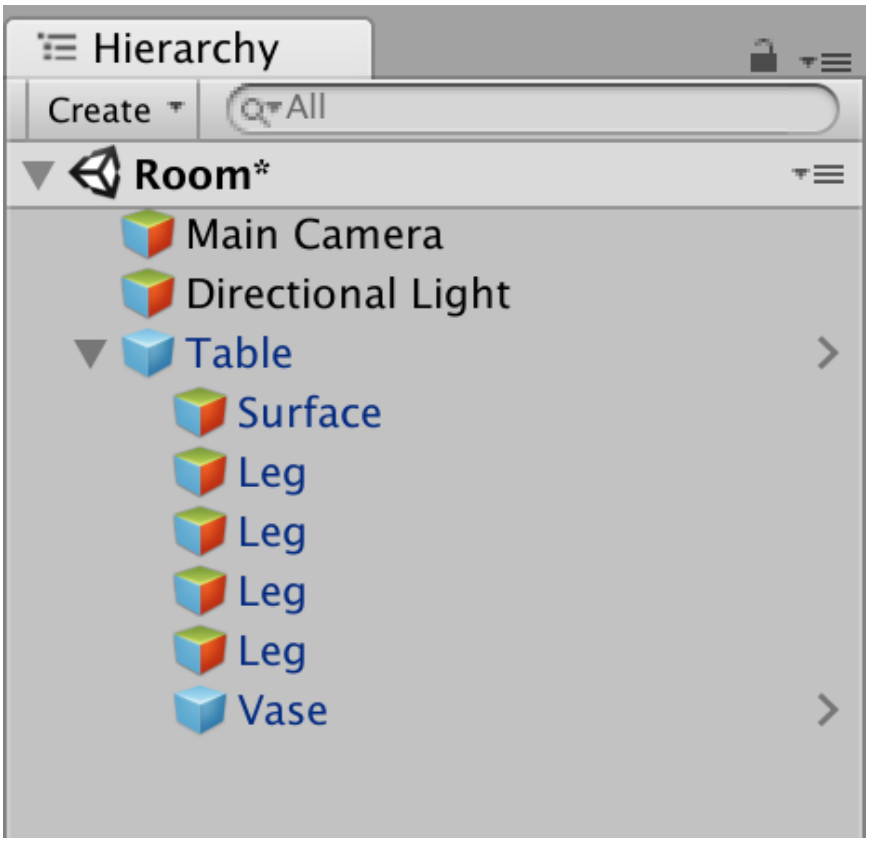
Leave feedback

When you work with Prefabs inside other Prefabs, or with **Prefab** Variants, overrides can exist at multiple levels, and the same overrides can have multiple different Prefabs they can be applied to.

Choice of apply target

When you have a Prefab instance which has nested Prefabs inside it, or which is a Prefab Variant, you might have a choice of which Prefab an override should be applied to.

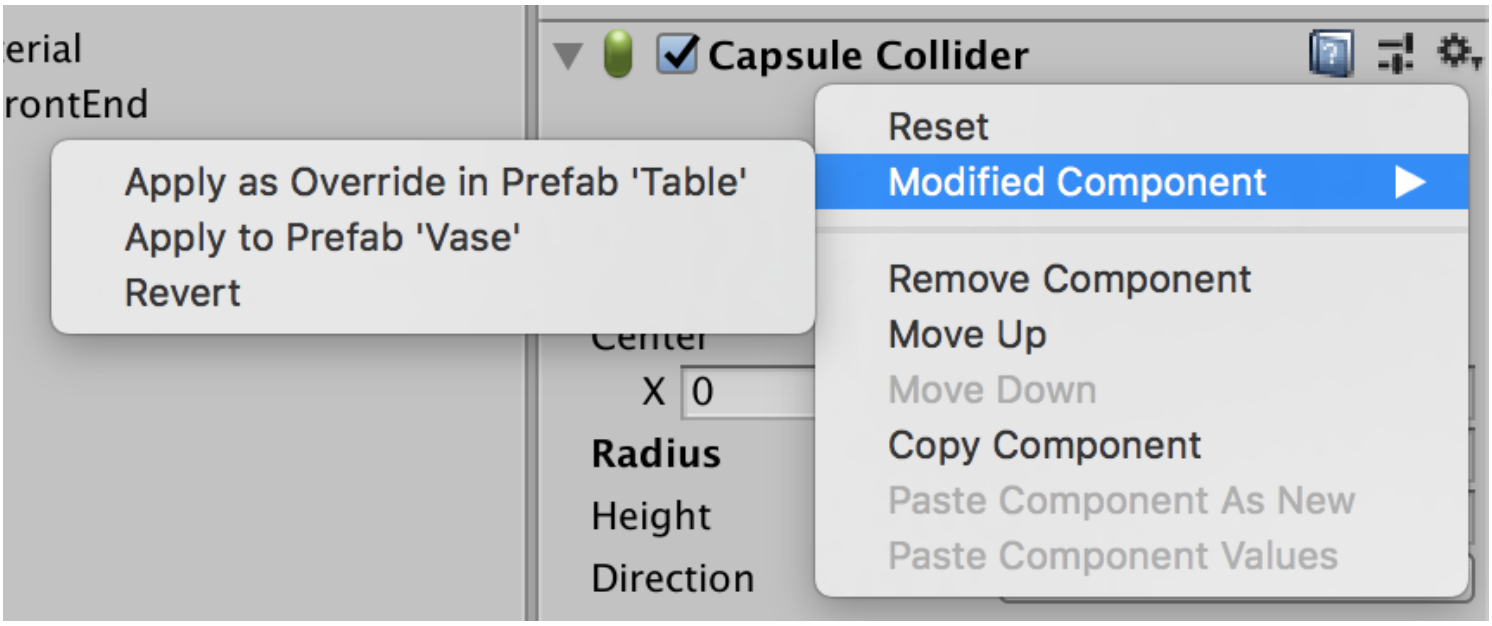
Consider a Prefab “Vase” which is nested inside a Prefab “Table”, and the **scene** contains an instance of the “Table” Prefab.



A ‘Vase’ Prefab nested inside a ‘Table’ Prefab.

If on this instance, a property on “Vase” is overridden, there are multiple Prefabs this override could be applied to: the “Vase” or the “Table”.

The **Apply All** button in the Overrides drop-down window only allows applying the override to the outer Prefab - the “Table” in this case. But a choice of apply target is available when applying either via the context menu, or via the comparison view for individual **components** in the Overrides drop-down window.



In this example, if you choose **Apply to Prefab 'Vase'**, the value is applied to the 'Vase' Prefab **Asset** and is used for all instances of the 'Vase' Prefab.

And, if you choose **Apply as Override in Prefab 'Table'**, the value becomes an override on the instance of 'Vase' that is inside the 'Table' Prefab. The property is no longer marked as an override on the instance in the Scene, but if you open the 'Table' Prefab in Prefab Mode, the property on the 'Vase' Prefab instance is marked as an override there.

The 'Vase' Prefab Asset itself is not affected at all when overriding as an override in the 'Table' Prefab Asset. This means that all instances of the 'Table' Prefab now have the new value on their 'Vase' Prefab instance, but other instances of the 'Vase' Prefab that are not part of the 'Table' Prefab are not affected.

If the property on the 'Vase' Prefab itself is later changed, it will affect all instances of the 'Vase' Prefab, except where that property is overridden. Since it's overridden on the 'Vase' instance inside the 'Table' Prefab, the change won't affect any of the 'Vase' instances that are part of 'Table' instances.

Applying to inner Prefabs may affect outer Prefabs too

Applying one or more properties to an inner Prefab Asset can sometimes modify outer Prefab Assets as well, because those properties get their overrides reverted in the outer Prefabs.

In our example, if **Apply to Prefab 'Vase'** is chosen and the 'Table' Prefab has an override of the value, this override in the 'Table' Prefab is reverted at the same time so that the property on the instance retains the value that was just applied. If this was not the case, the value on the instance would change right after being applied.

2018–07–31 Page published with limited editorial review

Nested Prefabs and Prefab Variants added in 2018.3

Unpacking Prefab instances

Leave feedback

To return the contents of a **Prefab** instance into a regular **GameObject**, you unpack the **Prefab** instance. This is exactly the reverse operation of creating (packing) a Prefab, except that it doesn't destroy the Prefab **Asset** but only affects the Prefab instance.

You can unpack a Prefab instance by right-clicking on it in the Hierarchy and selecting **Unpack Prefab**. The resulting GameObject in the **Scene** no longer has any link to its former Prefab Asset. The Prefab Asset itself is not affected by this operation and there may still be other instances of it in your Project.

If you had any overrides on your Prefab instance before you unpacked it, those will be "baked" onto the resulting GameObjects. That is, the values will stay the same, but will no longer have status as overrides, since there's no Prefab to override.

If you unpack a Prefab that has nested Prefabs inside, the nested Prefabs remain Prefab instances and still have links to their respective Prefab Assets. Similarly, if you unpack a Prefab Variant, there will be a new Prefab instance at the root which is an instance of the base Prefab.

In general, unpacking a Prefab instance will give you the same objects you see if you go into Prefab Mode for that Prefab. This is because Prefab Mode shows the contents that is inside of a Prefab, and unpacking a Prefab instance unpacks the contents of a Prefab.

If you have a Prefab instance that you want to replace with plain GameObjects and completely remove all links to any Prefab Assets, you can right-click on it in the Hierarchy and select **Unpack Prefab Completely**. This is equivalent to unpacking the Prefab, and keeping on unpacking any Prefab instances that appear as a result because they were nested Prefabs or base Prefabs.

You can unpack Prefab instances that exist in Scenes, or which exist inside other Prefabs.

2018–07–31 Page published with no editorial review

Nested Prefabs and Prefab Variants added in 2018.3

Instantiating Prefabs at runtime

Leave feedback

By this point you should understand the concept of **Prefabs** at a fundamental level. They are a collection of predefined **GameObjects** & **Components** that are re-usable throughout your game. If you don't know what a Prefab is, we recommend you read the Prefabs page for a more basic introduction.

Prefabs come in very handy when you want to instantiate complicated GameObjects at runtime. The alternative to instantiating Prefabs is to create GameObjects from scratch using code. Instantiating Prefabs has many advantages over the alternative approach:

You can instantiate a Prefab from one line of code, with complete functionality. Creating equivalent GameObjects from code takes an average of five lines of code, but likely more.
You can set up, test, and modify the Prefab quickly and easily in the **Scene** and **Inspector**.
You can change the Prefab being instanced without changing the code that instantiates it. A simple rocket might be altered into a super-charged rocket, and no code changes are required.

Common Scenarios

To illustrate the strength of Prefabs, let's consider some basic situations where they would come in handy:

Building a wall out of a single "brick" Prefab by creating it several times in different positions.
A rocket launcher instantiates a flying rocket Prefab when fired. The Prefab contains a **Mesh**, **Rigidbody**, **Collider**, and a child GameObject with its own trail **Particle System**.
A robot exploding to many pieces. The complete, operational robot is destroyed and replaced with a wrecked robot Prefab. This Prefab would consist of the robot split into many parts, all set up with Rigidbodies and Particle Systems of their own. This technique allows you to blow up a robot into many pieces, with just one line of code, replacing one object with a Prefab.

Building a wall

This explanation will illustrate the advantages of using a Prefab vs creating objects from code.

First, lets build a brick wall from code:

```
public class Instantiation : MonoBehaviour
{
    void Start()
    {
        for (int y = 0; y < 5; y++)
        {
            for (int x = 0; x < 5; x++)
            {
                GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
                cube.AddComponent<Rigidbody>();
                cube.transform.position = new Vector3(x, y, 0);
            }
        }
    }
}
```

To use the above script we simply save the script and drag it onto an empty GameObject.
Create an empty GameObject with **GameObject->Create Empty**.

If you execute that code, you will see an entire brick wall is created when you enter Play mode. There are two lines relevant to the functionality of each individual brick: the `CreatePrimitive` line, and the `AddComponent` line. Not so bad right now, but each of our bricks is un-textured. Every additional action to want to perform on the brick, like changing the texture, the friction, or the Rigidbody **mass**, is an extra line.

If you create a Prefab and perform all your setup before-hand, you use one line of code to perform the creation and setup of each brick. This relieves you from maintaining and changing a lot of code when you decide you want to make changes. With a Prefab, you just make your changes and Play. No code alterations required.

If you're using a Prefab for each individual brick, this is the code you need to create the wall.

```
//Instantiate accepts any component type, because it instantiates the GameObject

public Transform brick;

void Start()
{
    for (int y = 0; y < 5; y++)
    {
        for (int x = 0; x < 5; x++)
        {
            Instantiate(brick, new Vector3(x, y, 0), Quaternion.identity);
        }
    }
}
```

This is not only very clean but also very reusable. There is nothing saying we are instantiating a cube or that it must contain a rigidbody. All of this is defined in the Prefab and can be quickly created in the Editor.

Now we only need to create the Prefab, which we do in the Editor. Here's how:

Choose **GameObject > 3D Object > Cube**
Choose **Component > Physics > Rigidbody**
Choose **Assets > Create > Prefab**
In the **Project View**, change the name of your new Prefab to "Brick"
Drag the cube you created in the **Hierarchy** onto the "Brick" Prefab in the **Project View**
With the Prefab created, you can safely delete the Cube from the Hierarchy (**Delete** on Windows, **Command-Backspace** on Mac)
We've created our Brick Prefab, so now we have to attach it to the **brick** variable in our script. When you select the empty GameObject that contains the script, the Brick variable will be visible in the inspector.

Now drag the "Brick" Prefab from the Project View onto the **brick** variable in the Inspector. Press Play and you'll see the wall built using the Prefab.

This is a workflow pattern that can be used over and over again in Unity. In the beginning you might wonder why this is so much better, because the script creating the cube from code is only 2 lines longer.

But because you are using a Prefab now, you can adjust the Prefab in seconds. Want to change the mass of all those instances? Adjust the Rigidbody in the Prefab only once. Want to use a different **Material** for all the instances? Drag the Material onto the Prefab only once. Want to change friction? Use a different **Physic Material** in the Prefab's collider. Want to add a Particle System to all those boxes? Add a child to the Prefab only once.

Instantiating rockets & explosions

Here's how Prefabs fit into this scenario:

A rocket launcher instantiates a rocket Prefab when the user presses fire. The Prefab contains a mesh, Rigidbody, Collider, and a child GameObject that contains a trail particle system.
The rocket impacts and instantiates an explosion Prefab. The explosion Prefab contains a Particle System, a light that fades out over time, and a script that applies damage to surrounding GameObjects.
While it would be possible to build a rocket GameObject completely from code, adding Components manually and setting properties, it is far easier to instantiate a Prefab. You can instantiate the rocket in just one line of code, no matter how complex the rocket's Prefab is. After instantiating the Prefab you can also modify any properties of the instantiated object (e.g. you can set the velocity of the rocket's Rigidbody).

Aside from being easier to use, you can update the prefab later on. So if you are building a rocket, you don't immediately have to add a Particle trail to it. You can do that later. As soon as you add the trail as a child GameObject to the Prefab, all your instantiated rockets will have particle trails. And lastly, you can quickly tweak the properties of the rocket Prefab in the Inspector, making it far easier to fine-tune your game.

This script shows how to launch a rocket using the **Instantiate()** function.

```
// Require the rocket to be a rigidbody.
// This way we the user can not assign a prefab without rigidbody
public Rigidbody rocket;
public float speed = 10f;

void FireRocket ()
{
    Rigidbody rocketClone = (Rigidbody) Instantiate(rocket, transform.position, transform.ro
    rocketClone.velocity = transform.forward * speed;

    // You can also access other components / scripts of the clone
    rocketClone.GetComponent<MyRocketScript>().DoSomething();
}

// Calls the fire method when holding down ctrl or mouse
void Update ()
{
    if (Input.GetButtonDown("Fire1"))
    {
        FireRocket();
    }
}
```

Replacing a character with a ragdoll or wreck

Let's say you have a fully rigged enemy character who dies. You could simply play a death animation on the character and disable all **scripts** that usually handle the enemy logic. You probably have to take care of removing several **scripts**, adding some custom logic to make sure that no one will continue attacking the dead enemy anymore, and other cleanup tasks.

A far better approach is to immediately delete the entire character and replace it with an instantiated wrecked prefab. This gives you a lot of flexibility. You could use a different material for the dead character, attach completely different scripts, spawn a Prefab containing the object broken into many pieces to simulate a shattered enemy, or simply instantiate a Prefab containing a version of the character.

Any of these options can be achieved with a single call to **Instantiate()**, you just have to hook it up to the right prefab and you're set!

The important part to remember is that the wreck which you **Instantiate()** can be made of completely different objects than the original. For example, if you have an airplane, you would model two versions. One where the plane consists of a single GameObject with **Mesh Renderer** and scripts for airplane physics. By keeping the model in just one GameObject, your game will run faster since you will be able to make the model with less triangles and since it consists of fewer objects it will render faster than using many small parts. Also while your plane is happily flying around there is no reason to have it in separate parts.

To build a wrecked airplane Prefab, the typical steps are:

Model your airplane with lots of different parts in your favorite modeler
Create an empty Scene
Drag the model into the empty Scene
Add Rigidbodies to all parts, by selecting all the parts and choosing **Component->Physics->Rigidbody**
Add Box Colliders to all parts by selecting all the parts and choosing **Component->Physics->Box Collider**
For an extra special effect, add a smoke-like Particle System as a child GameObject to each of the parts
Now you have an airplane with multiple exploded parts, they fall to the ground by physics and will create a Particle trail due to the attached particle system. Hit Play to preview how your model reacts and do any necessary tweaks.
Choose **Assets->Create Prefab**
Drag the root GameObject containing all the airplane parts into the Prefab
The following example shows how these steps are modelled in code.

```
public GameObject wreck;

// As an example, we turn the game object into a wreck after 3 seconds automatically
IEnumerator Start()
{
    yield return new WaitForSeconds(3);
    KillSelf();
}

// Calls the fire method when holding down ctrl or mouse
void KillSelf ()
{
    // Instantiate the wreck game object at the same position we are at
    GameObject wreckClone = (GameObject) Instantiate(wreck, transform.position, transform.ro

    // Sometimes we need to carry over some variables from this object
    // to the wreck
    wreckClone.GetComponent<MyScript>().someVariable = GetComponent<MyScript>().someVariable

    // Kill ourselves
    Destroy(gameObject);
}
```

Placing a bunch of objects in a specific pattern

Lets say you want to place a bunch of objects in a grid or circle pattern. Traditionally this would be done by either:

Building an object completely from code. This is tedious! Entering values from a script is both slow, unintuitive and not worth the hassle.
Make the fully rigged object, duplicate it and place it multiple times in the scene. This is tedious, and placing objects accurately in a grid is hard.
So use **Instantiate()** with a Prefab instead! We think you get the idea of why Prefabs are so useful in these scenarios. Here's the code necessary for these scenarios:

```
// Instantiates a prefab in a circle

public GameObject prefab;
public int numberOfObjects = 20;
public float radius = 5f;

void Start()
{
    for (int i = 0; i < numberOfObjects; i++)
    {
        float angle = i * Mathf.PI * 2 / numberOfObjects;
        Vector3 pos = new Vector3(Mathf.Cos(angle), 0, Mathf.Sin(angle)) * radius;
        Instantiate(prefab, pos, Quaternion.identity);
    }
}
```

```
// Instantiates a prefab in a grid

public GameObject prefab;
public float gridX = 5f;
public float gridY = 5f;
public float spacing = 2f;

void Start()
{
    for (int y = 0; y < gridY; y++)
    {
        for (int x = 0; x < gridX; x++)
        {
            Vector3 pos = new Vector3(x, 0, y) * spacing;
            Instantiate(prefab, pos, Quaternion.identity);
        }
    }
}
```

Input

Leave feedback

Unity supports the most conventional types of input device used with games (such as controllers, joypads, keyboard and mouse) but also the touchscreens and movement-sensing capabilities of mobile devices. It also supports input from VR and **AR** systems (see XR input).

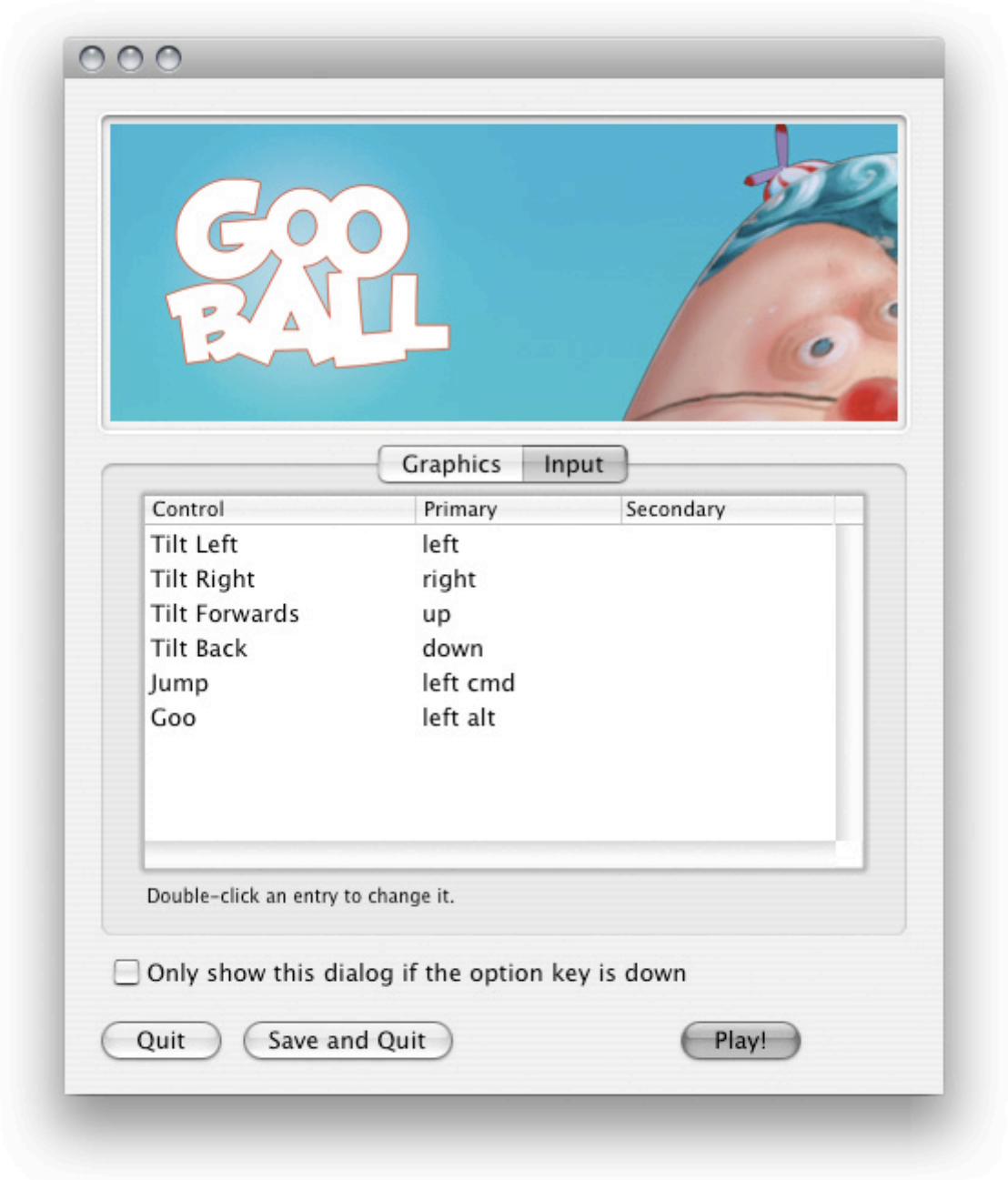
Additionally, Unity can make use of a computer's microphone and webcam to input audio and video data. See the Audio and Graphics sections of the manual for further information.

Conventional Game Input

Leave feedback

Unity supports keyboard, joystick and gamepad input.

Virtual axes and buttons can be created in the **Input Manager**, and end users can configure Keyboard input in a nice screen configuration dialog.



NOTE: This is a legacy image. This Input Selector image dates back to the very earliest versions of the Unity Editor in 2005. GooBall was a Unity Technologies game.

You can setup joysticks, gamepads, keyboard, and mouse, then access them all through one simple scripting interface. Typically you use the axes and buttons to fake up a console controller. Alternatively you can access keys on the keyboard.

Virtual Axes

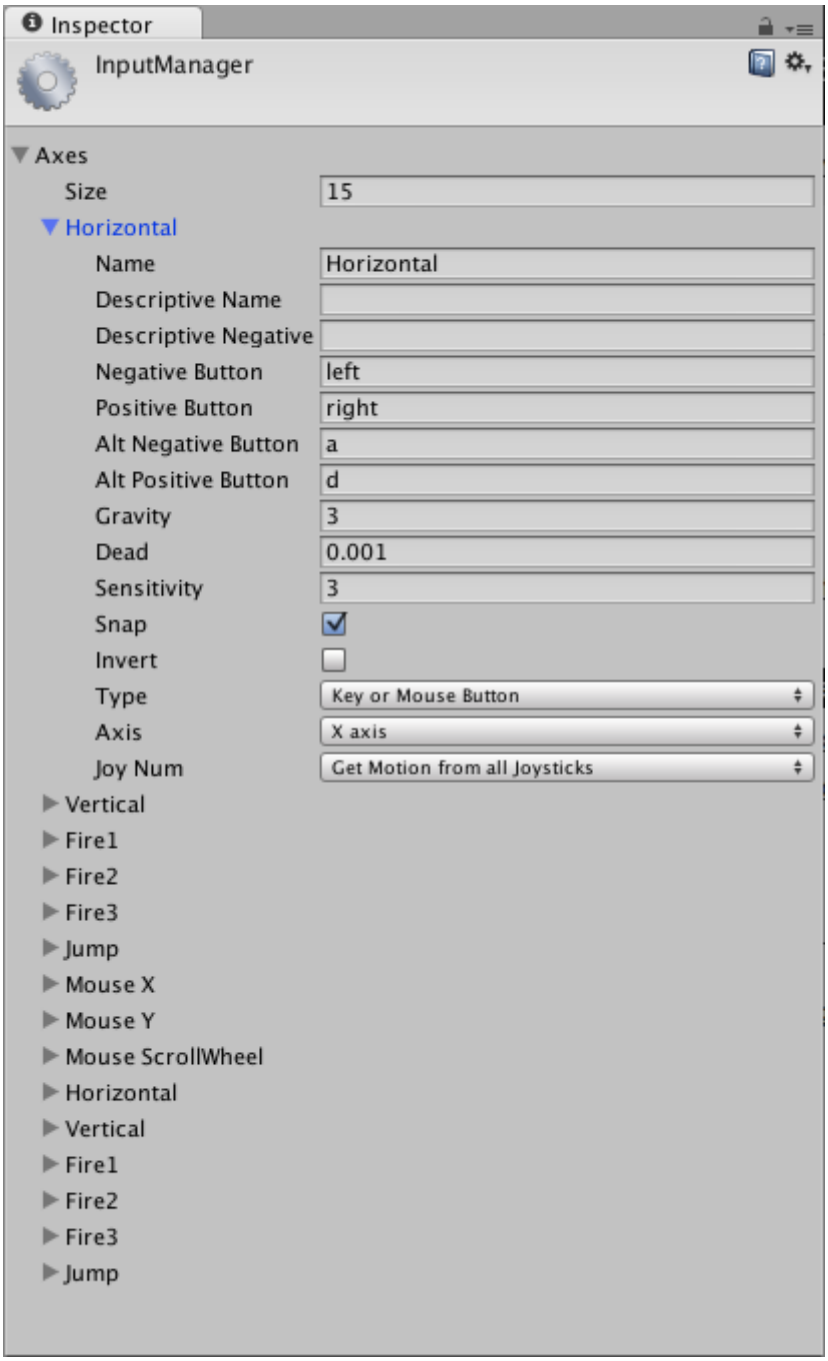
From **scripts**, all virtual axes are accessed by their name.

Every project has the following default input axes when it's created:

Horizontal and **Vertical** are mapped to w, a, s, d and the arrow keys.
Fire1, **Fire2**, **Fire3** are mapped to Control, Option (Alt), and Command, respectively.
Mouse X and **Mouse Y** are mapped to the delta of mouse movement.
Window Shake X and **Window Shake Y** is mapped to the movement of the window.

Adding new Input Axes

If you want to add new virtual axes go to the **Edit->Project Settings->Input** menu. Here you can also change the settings of each axis.



You map each axis to two buttons on a joystick, mouse, or keyboard keys.

Property:	*Function:*
Name	The name of the string used to check this axis from a script.

Property:	*Function:*
Descriptive Name	Positive value name displayed in the input tab of the **Configuration** dialog for standalone builds.
Descriptive Negative Name	Negative value name displayed in the Input tab of the **Configuration** dialog for standalone builds.
Negative Button	The button used to push the axis in the negative direction.
Positive Button	The button used to push the axis in the positive direction.
Alt Negative Button	Alternative button used to push the axis in the negative direction.
Alt Positive Button	Alternative button used to push the axis in the positive direction.
Gravity	Speed in units per second that the axis falls toward neutral when no buttons are pressed.
Dead	Size of the analog dead zone. All analog device values within this range result map to neutral.
Sensitivity	Speed in units per second that the axis will move toward the target value. This is for digital devices only.
Snap	If enabled, the axis value will reset to zero when pressing a button of the opposite direction.
Invert	If enabled, the **Negative Buttons** provide a positive value, and vice-versa.
Type	The type of inputs that will control this axis.
Axis	The axis of a connected device that will control this axis.
Joy Num	The connected Joystick that will control this axis.

Use these settings to fine tune the look and feel of input. They are all documented with tooltips in the Editor as well.

Using Input Axes from Scripts

You can query the current state from a script like this:

```
value = Input.GetAxis ("Horizontal");
```

An axis has a value between –1 and 1. The neutral position is 0. This is the case for joystick input and keyboard input.

However, Mouse Delta and Window Shake Delta are how much the mouse or window moved during the last frame. This means it can be larger than 1 or smaller than –1 when the user moves the mouse quickly.

It is possible to create multiple axes with the same name. When getting the input axis, the axis with the largest absolute value will be returned. This makes it possible to assign more than one input device to one axis name. For example, create one axis for keyboard input and one axis for joystick input with the same name. If the user is using the joystick, input will come from the joystick, otherwise input will come from the keyboard. This way you don't have to consider where the input comes from when writing scripts.

Button Names

To map a key to an axis, you have to enter the key's name in the **Positive Button** or **Negative Button** property in the **Inspector**.

Keys

The names of keys follow this convention:

Normal keys: "a", "b", "c" ...
Number keys: "1", "2", "3", ...
Arrow keys: "up", "down", "left", "right"
Keypad keys: "[1]", "[2]", "[3]", "[+]", "[equals]"
Modifier keys: "right shift", "left shift", "right ctrl", "left ctrl", "right alt", "left alt", "right cmd", "left cmd"
Mouse Buttons: "mouse 0", "mouse 1", "mouse 2", ...
Joystick Buttons (from any joystick): "joystick button 0", "joystick button 1", "joystick button 2", ...
Joystick Buttons (from a specific joystick): "joystick 1 button 0", "joystick 1 button 1", "joystick 2 button 0", ...
Special keys: "backspace", "tab", "return", "escape", "space", "delete", "enter", "insert", "home", "end", "page up", "page down"
Function keys: "f1", "f2", "f3", ...
The names used to identify the keys are the same in the scripting interface and the Inspector.

```
value = Input.GetKey ("a");
```

Note also that the keys are accessible using the KeyCode enum parameter.

Mobile Device Input

Leave feedback

On mobile devices, the Input class offers access to touchscreen, accelerometer and geographical/location input.

Access to keyboard on mobile devices is provided via the iOS keyboard.

Multi-Touch Screen

The iPhone and iPod Touch devices are capable of tracking up to five fingers touching the screen simultaneously. You can retrieve the status of each finger touching the screen during the last frame by accessing the Input.touches property array.

Android devices don't have a unified limit on how many fingers they track. Instead, it varies from device to device and can be anything from two-touch on older devices to five fingers on some newer devices.

Each finger touch is represented by an Input.Touch data structure:

Property:	*Function:*
fingerId	The unique index for a touch.
position	The screen position of the touch.
deltaPosition	The screen position change since the last frame.
deltaTime	Amount of time that has passed since the last state change.
tapCount	The iPhone/iPad screen is able to distinguish quick finger taps by the user. This counter will let you know how many times the user has tapped the screen without moving a finger to the sides. Android devices do not count number of taps, this field is always 1.
phase	Describes so called "phase" or the state of the touch. It can help you determine if the touch just began, if user moved the finger or if they just lifted the finger.

Phase can be one of the following:

Began	A finger just touched the screen.
Moved	A finger moved on the screen.
Stationary	A finger is touching the screen but hasn't moved since the last frame.
Ended	A finger was lifted from the screen. This is the final phase of a touch.
Canceled	The system cancelled tracking for the touch, as when (for example) the user puts the device to their face or more than five touches happened simultaneously. This is the final phase of a touch.

Following is an example script which will shoot a ray whenever the user taps on the screen:

```
var particle : GameObject;
function Update () {
    for (var touch : Touch in Input.touches) {
        if (touch.phase == TouchPhase.Began) {
            // Construct a ray from the current touch coordinates
            var ray = Camera.main.ScreenPointToRay (touch.position);
            if (Physics.Raycast (ray)) {
```

```
                // Create a particle if hit
                Instantiate (particle, transform.position, transform.rotation);
            }
        }
    }
}
```

Mouse Simulation

On top of native touch support Unity **iOS**/Android provides a mouse simulation. You can use mouse functionality from the standard Input class. Note that iOS/Android devices are designed to support multiple finger touch. Using the mouse functionality will support just a single finger touch. Also, finger touch on mobile devices can move from one area to another with no movement between them. Mouse simulation on mobile devices will provide movement, so is very different compared to touch input. The recommendation is to use the mouse simulation during early development but to use touch input as soon as possible.

Accelerometer

As the mobile device moves, a built-in accelerometer reports linear acceleration changes along the three primary axes in three-dimensional space. Acceleration along each axis is reported directly by the hardware as G-force values. A value of 1.0 represents a load of about +1g along a given axis while a value of –1.0 represents –1g. If you hold the device upright (with the home button at the bottom) in front of you, the X axis is positive along the right, the Y axis is positive directly up, and the Z axis is positive pointing toward you.

You can retrieve the accelerometer value by accessing the Input.acceleration property.

The following is an example script which will move an object using the accelerometer:

```
var speed = 10.0;
function Update () {
    var dir : Vector3 = Vector3.zero;

    // we assume that the device is held parallel to the ground
    // and the Home button is in the right hand

    // remap the device acceleration axis to game coordinates:
    // 1) XY plane of the device is mapped onto XZ plane
    // 2) rotated 90 degrees around Y axis
    dir.x = -Input.acceleration.y;
    dir.z = Input.acceleration.x;

    // clamp acceleration vector to the unit sphere
    if (dir.sqrMagnitude > 1)
```

```
        dir.Normalize();

    // Make it move 10 meters per second instead of 10 meters per frame...
    dir *= Time.deltaTime;

    // Move object
    transform.Translate (dir * speed);
}
```

Low-Pass Filter

Accelerometer readings can be jerky and noisy. Applying low-pass filtering on the signal allows you to smooth it and get rid of high frequency noise.

The following script shows you how to apply low-pass filtering to accelerometer readings:

```
var AccelerometerUpdateInterval : float = 1.0 / 60.0;
var LowPassKernelWidthInSeconds : float = 1.0;

private var LowPassFilterFactor : float = AccelerometerUpdateInterval / LowPassK
private var lowPassValue : Vector3 = Vector3.zero;
function Start () {
    lowPassValue = Input.acceleration;
}

function LowPassFilterAccelerometer() : Vector3 {
    lowPassValue = Vector3.Lerp(lowPassValue, Input.acceleration, LowPassFilterF
    return lowPassValue;
}
```

The greater the value of `LowPassKernelWidthInSeconds`, the slower the filtered value will converge towards the current input sample (and vice versa).

I'd like as much precision as possible when reading the accelerometer. What should I do?

Reading the Input.acceleration variable does not equal sampling the hardware. Put simply, Unity samples the hardware at a frequency of 60Hz and stores the result into the variable. In reality, things are a little bit more

complicated – accelerometer sampling doesn't occur at consistent time intervals, if under significant CPU loads. As a result, the system might report 2 samples during one frame, then 1 sample during the next frame.

You can access all measurements executed by accelerometer during the frame. The following code will illustrate a simple average of all the accelerometer events that were collected within the last frame:

```
var period : float = 0.0;
var acc : Vector3 = Vector3.zero;
for (var evnt : iPhoneAccelerationEvent in iPhoneInput.accelerationEvents) {
    acc += evnt.acceleration * evnt.deltaTime;
    period += evnt.deltaTime;
}
if (period > 0)
    acc *= 1.0/period;
return acc;
```

Mobile Keyboard

Leave feedback

In most cases, Unity will handle keyboard input automatically for GUI elements but it is also easy to show the keyboard on demand from a script.

GUI Elements

The keyboard will appear automatically when a user taps on editable GUI elements. Currently, GUI.TextField, GUI.TextArea and GUI.PasswordField will display the keyboard; see the GUI class documentation for further details.

Manual Keyboard Handling

Use the **TouchScreenKeyboard.Open()** function to open the keyboard. Please see the TouchScreenKeyboard scripting reference for the parameters that this function takes.

Keyboard Layout Options

The Keyboard supports the following options:-

Property:	*Function:*
TouchScreenKeyboardType.Default	Letters. Can be switched to keyboard with numbers and punctuation.
TouchScreenKeyboardType.ASCIICapable	Letters. Can be switched to keyboard with numbers and punctuation.
TouchScreenKeyboardType.NumbersAndPunctuation	Numbers and punctuation. Can be switched to keyboard with letters.
TouchScreenKeyboardType.URL	Letters with slash and .com buttons. Can be switched to keyboard with numbers and punctuation.
TouchScreenKeyboardType.NumberPad	Only numbers from 0 to 9.
TouchScreenKeyboardType.PhonePad	Keyboard used to enter phone numbers.
TouchScreenKeyboardType.NamePhonePad	Letters. Can be switched to phone keyboard.
TouchScreenKeyboardType.EmailAddress	Letters with @ sign. Can be switched to keyboard with numbers and punctuation.

Text Preview

By default, an edit box will be created and placed on top of the keyboard after it appears. This works as preview of the text that user is typing, so the text is always visible for the user. However, you can disable text preview by setting **TouchScreenKeyboard.hideInput** to true. Note that this works only for certain keyboard types and input modes. For example, it will not work for phone keypads and multi-line text input. In such cases, the edit box will always appear. **TouchScreenKeyboard.hideInput** is a global variable and will affect all keyboards.

Visibility and Keyboard Size

There are three keyboard properties in TouchScreenKeyboard that determine keyboard visibility status and size on the screen.

Property: Function:

visible	Returns **true** if the keyboard is fully visible on the screen and can be used to enter characters.
area	Returns the position and dimensions of the keyboard.
active	Returns **true** if the keyboard is activated. This property is not static property. You must have a keyboard instance to use this property.

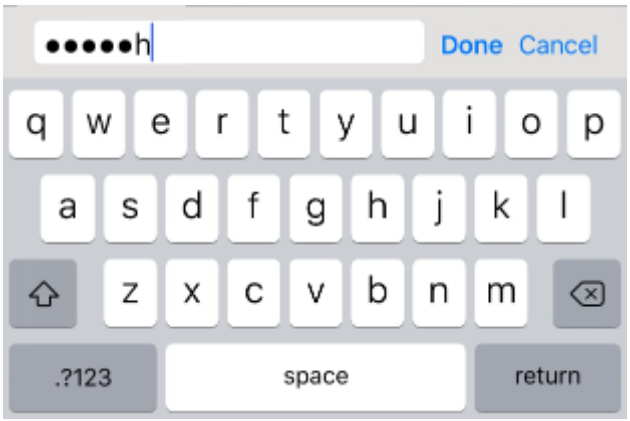
Note that **TouchScreenKeyboard.area** will return a Rect with position and size set to 0 until the keyboard is fully visible on the screen. You should not query this value immediately after **TouchScreenKeyboard.Open()**. The sequence of keyboard events is as follows:

TouchScreenKeyboard.Open() is called. **TouchScreenKeyboard.active** returns true. **TouchScreenKeyboard.visible** returns false. **TouchScreenKeyboard.area** returns (0, 0, 0, 0).
Keyboard slides out into the screen. All properties remain the same.
Keyboard stops sliding. **TouchScreenKeyboard.active** returns true. **TouchScreenKeyboard.visible** returns true. **TouchScreenKeyboard.area** returns real position and size of the keyboard.

Secure Text Input

It is possible to configure the keyboard to hide symbols when typing. This is useful when users are required to enter sensitive information (such as passwords). To manually open keyboard with secure text input enabled, use the following code:

```
TouchScreenKeyboard.Open("", TouchScreenKeyboardType.Default, false, false, true
```

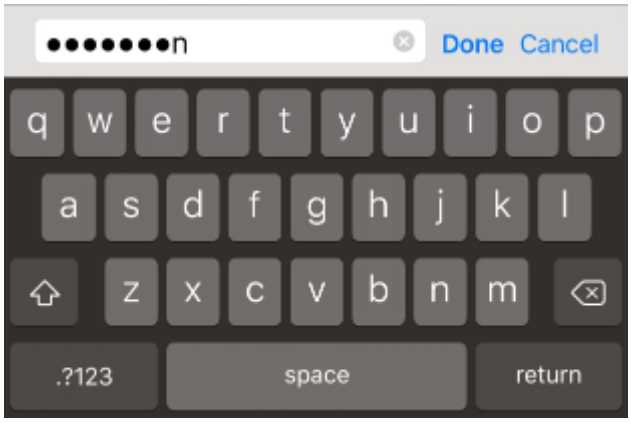


Hiding text while typing

Alert keyboard

To display the keyboard with a black semi-transparent background instead of the classic opaque, call **TouchScreenKeyboard.Open()** as follows:

```
TouchScreenKeyboard.Open("", TouchScreenKeyboardType.Default, false, false, true
```



Alert keyboard

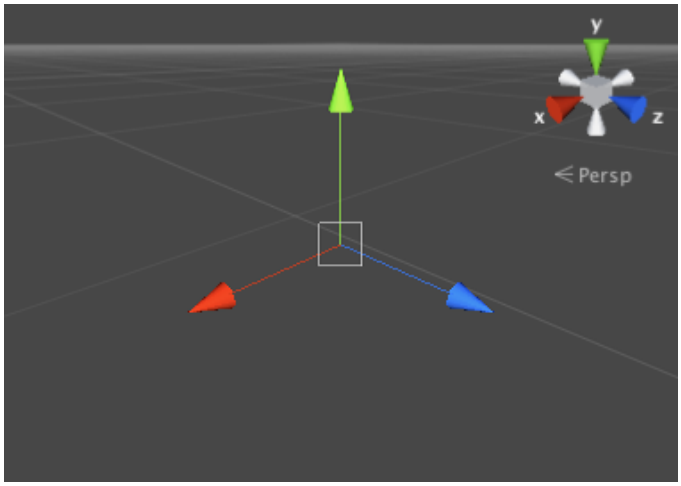
Transforms

Leave feedback

The **Transform** is used to store a **GameObject**'s position, rotation, scale and parenting state and is thus very important. A **GameObject** will always have a Transform **component** attached - it is not possible to remove a Transform or to create a **GameObject** without one.

Editing Transforms

Transforms are manipulated in 3D space in the X, Y, and Z axes or in 2D space in just X and Y. In Unity, these axes are represented by the colors red, green, and blue respectively.

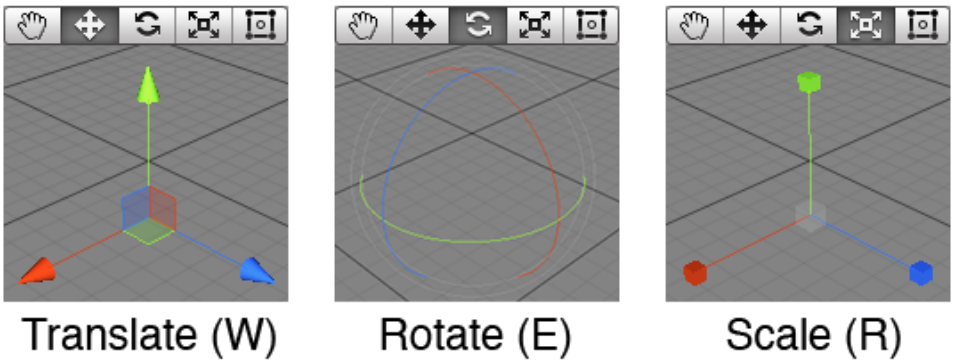


A transform showing the color-coding of the axes

A Transform can be edited in the **Scene View** or by changing its properties in the **Inspector**. In the **scene**, you can modify Transforms using the Move, Rotate and Scale tools. These tools are located in the upper left-hand corner of the Unity Editor.

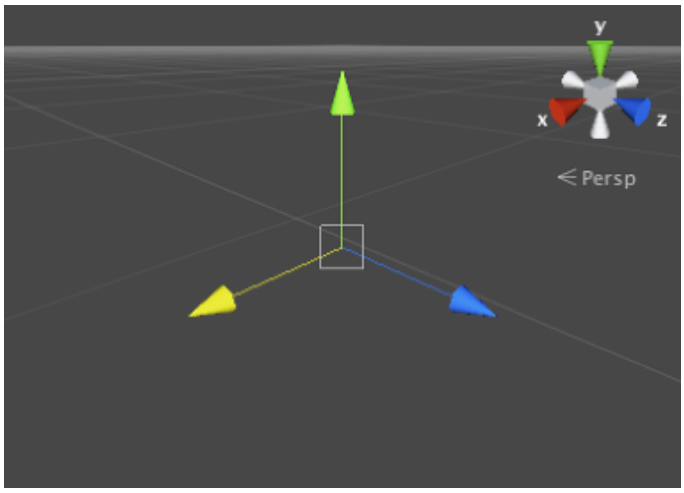
The View, Translate, Rotate, and Scale tools

The tools can be used on any object in the scene. When you click on an object, you will see the tool **gizmo** appear within it. The appearance of the **gizmo** depends on which tool is selected.



Transform gizmo

When you click and drag on one of the three gizmo axes, you will notice that its color changes to yellow. As you drag the mouse, you will see the object translate, rotate, or scale along the selected axis. When you release the mouse button, the axis remains selected.



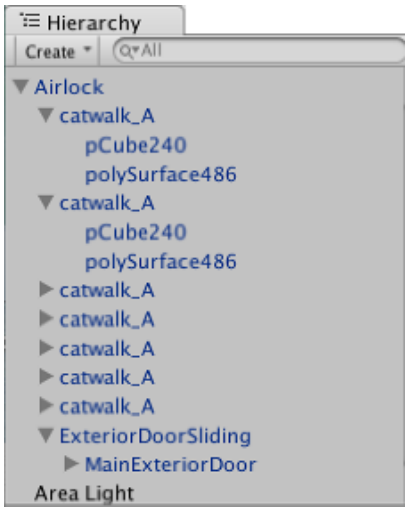
A Transform showing the selected (yellow) X axis

There is also an additional option in Translate mode to lock movement to a particular plane (ie, allow dragging in two of the axes while keeping the third unchanged). The three small coloured squares around the center of the Translate gizmo activate the lock for each plane; the colors correspond to the axis that will be locked when the square is clicked (eg, blue locks the Z axis).

Parenting

Parenting is one of the most important concepts to understand when using Unity. When a GameObject is a **Parent** of another GameObject, the **Child** GameObject will move, rotate, and scale exactly as its Parent does. You can think of parenting as being like the relationship between your arms and your body; whenever your body moves, your arms also move along with it. Child objects can also have children of their own and so on. So your hands could be regarded as "children" of your arms and then each hand has several fingers, **etc**. Any object can have multiple children, but only one parent. These multiple levels of parent-child relationships form a Transform *hierarchy*. The object at the very top of a hierarchy (ie, the only object in the hierarchy that doesn't have a parent) is known as the **root**.

You can create a Parent by dragging any GameObject in the **Hierarchy View** onto another. This will create a Parent-Child relationship between the two GameObjects.



Example of a Parent-Child hierarchy. GameObjects with foldout arrows to the left of their names are parents.

Note that the Transform values in the Inspector for any child GameObject are displayed relative to the Parent's Transform values. These values are referred to as **local coordinates**. Returning to the analogy of body and arms, the position of your body may move as you walk but your arms will still be attached at the same relative position. For scene construction, it is usually sufficient to work with local coordinates for child objects but in gameplay it is often useful to find their exact position in world space or **global coordinates**. The scripting API for the **Transform component** has separate properties for local and global position, rotation and scale and also allows you to convert any point between local and global coordinates.

Limitations with Non-Uniform Scaling

Non-uniform scaling is when the **Scale** in a Transform has different values for x, y, and z; for example (2, 4, 2). In contrast, uniform scaling has the same value for x, y, and z; for example (3, 3, 3). Non-uniform scaling can be useful in a few specific cases but it introduces a few oddities that don't occur with uniform scaling:-

Certain components do not fully support non-uniform scaling. For example, some components have a circular or spherical element defined by a **radius** property, among them **Sphere Collider**, **Capsule Collider**, **Light** and **Audio Source**. In cases like this the circular shape will not become elliptical under non-uniform scaling as you would expect and will simply remain circular.
When a child object has a non-uniformly scaled parent and is rotated relative to that parent, it may appear skewed or "sheared". There are components that support simple non-uniform scaling but don't work correctly when skewed like this. For example, a skewed **Box Collider** will not match the shape of the rendered **mesh** accurately.
For performance reasons, a child object of a non-uniformly scaled parent will not have its scale automatically updated when it rotates. As a result, the child's shape may appear to change abruptly when the scale eventually *is* updated, say if the child object is detached from the parent.

Importance of Scale

The scale of the Transform determines the difference between the size of a mesh in your modeling application and the size of that mesh in Unity. The mesh's size in Unity (and therefore the Transform's scale) is *very* important, especially during physics simulation. By default, the **physics engine** assumes that one unit in world space corresponds to one metre. If an object is very large, it can appear to fall in "slow motion"; the simulation is actually correct since effectively, you are watching a very large object falling a great distance.

There are three factors that can affect the scale of your object:

The size of your mesh in your 3D modeling application.
The **Mesh Scale Factor** setting in the object's **Import Settings**.
The **Scale** values of your Transform Component.
Ideally, you should not adjust the **Scale** of your object in the Transform Component. The best option is to create your models at real-life scale so you won't have to change your Transform's scale. The next best option is to adjust the scale at which your mesh is imported in the **Import Settings** for your individual mesh. Certain optimizations occur based on the import size, and instantiating an object that has an adjusted scale value can decrease performance. For more information, see the section about optimizing scale on the Rigidbody component reference page.

Tips for Working with Transforms

When parenting Transforms, it is useful to set the parent's location to <0,0,0> before adding the child. This means that the local coordinates for the child will be the same as global coordinates making it easier to be sure you have the child in the right position.
Particle Systems are not affected by the Transform's **Scale**. In order to scale a Particle System, you need to modify the properties in the System's Particle Emitter, Animator and Renderer.
If you are using **Rigidbodies** for physics simulation then be sure to read about the Scale property on the Rigidbody component reference page.
You can change the colors of the Transform axes (and other UI elements) from the preferences (**Menu: Unity > Preferences** and then select the **Colors & keys** panel).
Changing the Scale affects the position of child transforms. For example scaling the parent to (0,0,0) will position all children at (0,0,0) relative to the parent.

Constraints

Leave feedback

A Constraint **component** links the position, rotation, or scale of a **GameObject** to another **GameObject**. A constrained **GameObject** moves, rotates, or scales like the **GameObject** it is linked to.

Unity supports the following types of Constraint components:

Aim: Rotate the constrained GameObject to face the linked GameObject.

Parent: Move and rotate the constrained GameObject with the linked GameObject.

Position: Move the constrained GameObject like the linked GameObject.

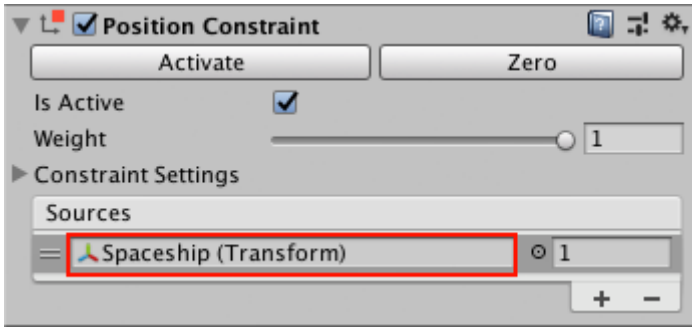
Rotation: Rotate the constrained GameObject like the linked GameObject.

Scale: Scale the constrained GameObject like the linked GameObject.

Linking to GameObjects

Use the **Sources** list in a Constraint component to specify the GameObjects to link to.

For example, to make a crosshair follow the player's spaceship in a 2D shooter, add a **Position Constraint** component to the crosshair. To link the crosshair to the spaceship, navigate to the **Position Constraint** component and add the spaceship GameObject to the **Sources** list. As the player moves the spaceship, the crosshair follows.



A Position Constraint for a crosshair. The crosshair follows the player's spaceship (red).

A Constraint can link to several source GameObjects. In this case, the Constraint uses the averaged position, rotation, or scale of its source GameObjects. For example, to point a Light at a group of GameObjects, add an **Aim Constraint** component to the Light GameObject. Then, add the GameObjects to illuminate in the **Sources** list. The **Aim Constraint** orients the light to face the averaged position of its sources.

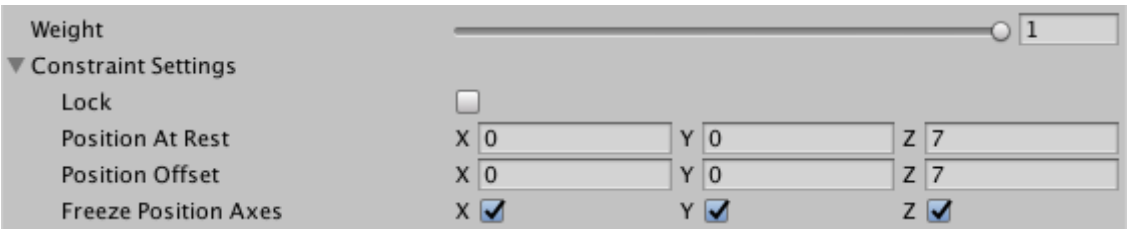
Unity evaluates source GameObjects in the order that they appear in the **Sources** list. The order has no effect for the **Position** and **Scale Constraints**. However, order has an effect on **Parent**, **Rotation**, and **Aim Constraints**. To get the result you want, reorder the **Sources** list by dragging and dropping items.

You can constrain a series of GameObjects. For example, you want ducklings to follow their mother in a row. You add a **Position Constraint** component to GameObject *Duckling1*. In the **Sources** list, you link to *MotherDuck*. You then add a **Position Constraint** to *Duckling2* that links to *Duckling1*. As the *MotherDuck* GameObject moves in the **Scene**, *Duckling1* follows *MotherDuck*, and *Duckling2* follows *Duckling1*.

Avoid creating a cycle of Constraints, because this causes unpredictable updates during gameplay.

Setting Constraint properties

Use the Inspector window to change common properties in a Constraint.



Weight and Constraint Settings for a Position Constraint

Use **Weight** to vary the influence of a Constraint. A weight of 1 causes the Constraint to update a GameObject at the same rate as its source GameObjects. A weight of 0 removes the effect of the Constraint completely. Each source GameObject also has an individual weight.

In **Constraint Settings**, use the **At Rest** properties to specify the X, Y, and Z values to use when **Weight** is 0 or when the corresponding property in **Freeze Axes** is not checked.

Use the **Offset** properties in **Constraint Settings** to specify the X, Y, and Z values to use when constraining the GameObject.

Use the **Freeze Axes** settings to toggle which axes the Constraint can actually modify.

Activating and locking Constraints

There are two aspects to working with Constraints: activating and locking.

You activate a Constraint to allow it to evaluate the position, rotation, or scale of the constrained GameObject. Unity does not evaluate inactive Constraints.

You lock a Constraint to allow it to move, rotate, or scale the GameObject. A locked Constraint takes control of the relevant parts of the Transform of the GameObject. You cannot manually move, rotate, or scale a GameObject with a locked Constraint. You also cannot edit the **Constraint Settings**.

To manually edit the position, rotation, or scale of a GameObject, unlock its Constraint. If the Constraint is active while unlocked, the Constraint updates **Constraint Settings** for you as you move, rotate, or scale the constrained GameObject or its source GameObjects.

When you add a Constraint component to a GameObject, the Constraint is inactive and unlocked by default. This lets you fine-tune the position, rotation, and scale of the constrained and source GameObjects before you activate and lock the Constraint.

For convenience, the **Activate** and **Zero** buttons update **Constraint Settings** for you:

Activate: Saves the current offset from the source GameObjects, then activates and locks the constrained GameObject**.**

Zero: Resets the position, rotation, or scale to match the source GameObjects, then activates and locks the constrained GameObject.

Animating and combining Constraints

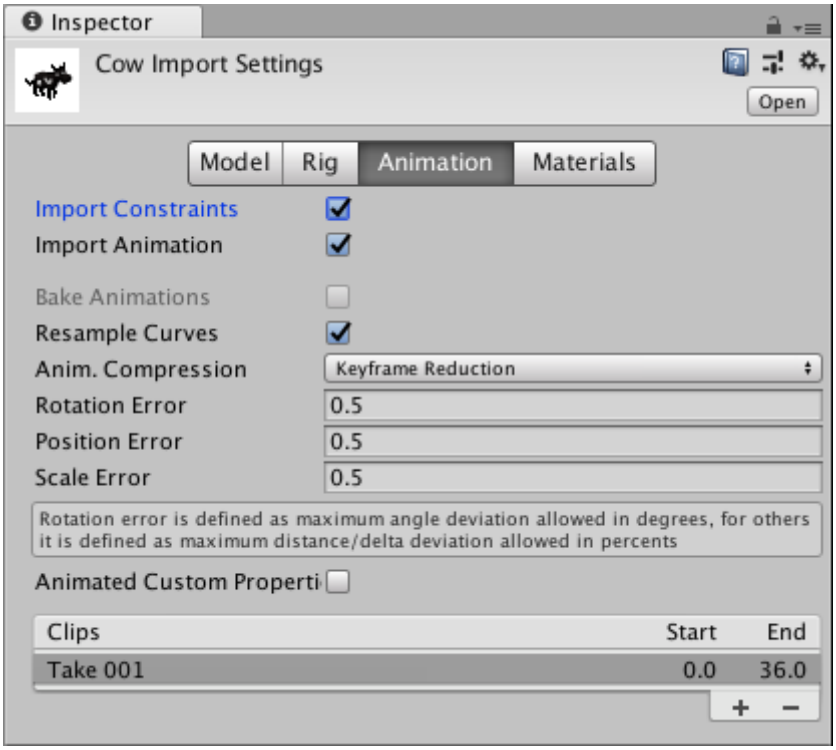
Use **animation clips** to modify the source GameObjects that your constrained GameObject links to. As the animation modifies the source GameObjects, the Constraint modifies your constrained GameObject.

You can also animate properties in a Constraint component. For example, use a Parent Constraint to move a character's sword from their hand to their back. First, add a Parent Constraint to the sword GameObject. In the **Sources** list, link the Constraint to the character's hand and the character's spine. To animate the sword, add **keyframes** for the weight of each source. To animate the sword moving from back to hand, add **keyframes** to change the weight of the hand from 0 to 1 with **keyframes** for the weight of the spine from 1 to 0.

You can add more than one kind of Constraint component to the same GameObject. When updating the GameObject, Unity evaluates Constraint components from first to last as they appear in the Inspector window. A GameObject can only contain one Constraint component of the same kind. For example, you cannot add more than one Position Constraint.

Importing Constraints

When importing FBX files into the Unity Editor from Autodesk® Maya® and MotionBuilder®, you can include Constraints. Click the Animation tab of the Import Settings window and check **Import Constraints**:



Import Settings with Import Constraints checked

For every constraint in the FBX file, Unity automatically adds a corresponding Constraint component and links it to the correct GameObjects.

Adding and editing Constraints

To add a Constraint component:

Select the GameObject to constrain.

In the **Inspector** window, click **Add Component** search for the type of Constraint you want to add, and click it to add it.

To add a source GameObject to your new Constraint, drag that GameObject from the Hierarchy (or from the Scene view) into the **Sources** list.

Move, rotate, or scale the constrained GameObject and its source GameObjects.

To activate the Constraint, click **Activate** or **Zero**, or check **Is Active** and **Lock**.

To edit a Constraint component:

Select the constrained GameObject in the Editor.

To adjust the **At Rest** or **Offset** fields, use the Inspector window to expand **Constraint Settings**, uncheck **Lock**, then edit the values.

To specify the axes that the Constraint updates, expand **Constraint Settings**, then check the properties in **Freeze Axes**.

To add a source GameObject to the Constraint:

If there are no empty slots in the Sources list, click + at the bottom of the list.

Drag the GameObject you want to use as a Constraint source from your scene into the **Sources** list.

To remove a source GameObject, select it in the **Sources** list and click the minus symbol (**-**) at the bottom of the list.

To re-order the source GameObjects in the **Sources** list, click the double bar icon on the left side of each GameObject you want to move, and drag it up or down.

Note: In the **Sources** list, order has no effect on the Position, Rotation, and Scale Constraints. However, order does affect how Parent and Aim Constraints move or rotate a GameObject.

Check **Is Active** and **Lock**.

2018–04–11 Page published with editorial review

Constraints added in 2018.1

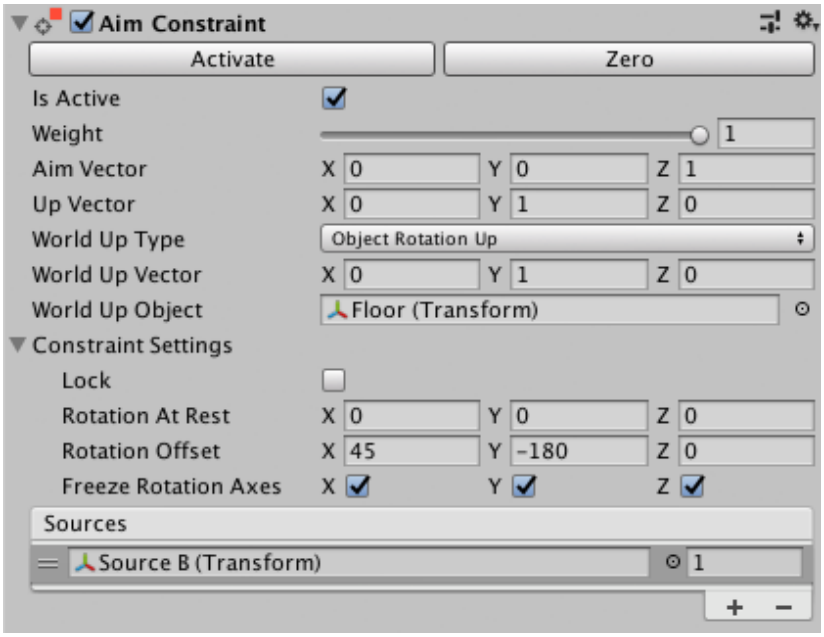
Aim Constraints

Leave feedback

An Aim Constraint rotates a GameObject to face its source **GameObjects**.

At the same time the Aim Constraint rotates a GameObject to follow its source GameObjects, it can also maintain a consistent orientation for another axis. For example, you add an Aim Constraint to a **Camera**. To keep the **Camera** upright while the Constraint aims it, specify the up axis of the **Camera** and an up direction to align it to.

Use the **Up Vector** to specify the up axis of the constrained GameObject. Use the **World Up Vector** to specify the upward direction. As the Aim Constraint rotates the GameObject to face its source GameObjects, the Constraint also aligns the up axis of the constrained GameObject with the upward direction.



Aim Constraint component

Properties

Property:	Function:
Activate	After you rotate the constrained GameObject and move its source GameObjects, click **Activate** to save this information. **Activate** saves the current offset from the source GameObjects in **Rotation At Rest** and **Rotation Offset**, then checks **Is Active** and **Lock**.
Zero	Sets the rotation of the constrained GameObject to the source GameObjects. Zero resets the **Rotation At Rest** and **Rotation Offset** fields, then checks **Is Active** and **Lock**.
Is Active	Toggles whether or not to evaluate the Constraint. To also apply the Constraint, make sure **Lock** is checked.
Weight	The strength of the Constraint. A weight of 1 causes the Constraint to rotate this GameObject at the same rate that its source GameObjects move. A weight of 0 removes the effect of the Constraint completely. This weight affects all source GameObjects. Each GameObject in the **Sources** list also has a weight.
Aim Vector	Specifies the axis which faces the direction of its source GameObjects. For example, to specify that the GameObject should orient only its positive Z axis to face the source GameObjects, enter an **Aim Vector** of 0, 0, 1 for the X, Y, and Z axes, respectively.

Property:	Function:
Up Vector	Specifies the up axis of this GameObject. For example, to specify that the GameObject should always keep its positive Y axis pointing upward, enter an **Up Vector** of 0, 1, 0 for the X, Y, and Z axes, respectively.
World Up Type	Specifies the axis for the upward direction. The Aim Constraint uses this vector to align the up axis of the GameObject the upward direction.
Scene Up	The Y axis of the **scene**.
Object Up	The Y axis of the GameObject referred to by **World Up Object**.
Object Up Rotation	The axis specified by **World Up Vector** of the GameObject referred to by **World Up Object**.
Vector	The World Up Vector.
None	Do not use a World Up vector.
World Up Vector	Specifies the vector to use for the **Object Up Rotation** and **Vector** choices in **World Up Type**.
World Up Object	Specifies the GameObject to use for the **Object Up** and **Object Up Rotation** choices in **World Up Type**.
Constraint Settings	
Lock	Toggle to let the Constraint rotate the GameObject. Uncheck this property to edit the rotation of this GameObject. You can also edit the Rotation At Rest and Rotation Offset properties. If Is Active is checked, the Constraint updates the Rotation At Rest or Rotation Offset properties for you as you rotate the GameObject or its source GameObjects. When you are satisfied with your changes, check Lock to let the Constraint control this GameObject. This property has no effect in Play Mode.
Rotation At Rest	The X, Y, and Z values to use when Weight is 0 or when the corresponding Freeze Rotation Axes are not checked. To edit these fields, uncheck Lock.
Rotation Offset	The X, Y, and Z offset from the rotation that is calculated by the Constraint. To edit these fields, uncheck Lock.
Freeze Rotation Axes	Check X, Y, or Z to allow the Constraint to control the corresponding axes. Uncheck an axis to stop the Constraint from controlling it. This allows you to edit, animate, or script the unfrozen axis.
Sources	The list of GameObjects that constrain this GameObject. Unity evaluates source GameObjects in the order that they appear in this list. This order affects how this Constraint rotates the constrained GameObject. To get the result you want, drag and drop items in this list. Each source has a weight from 0 to 1.

2018–03–13 Page published with editorial review

Constraints added in 2018.1

Parent Constraints

Leave feedback

A Parent Constraint moves and rotates a **GameObject** as if it is the child of another **GameObject** in the Hierarchy window. However, it offers certain advantages that are not possible when you make one **GameObject** the parent of another:

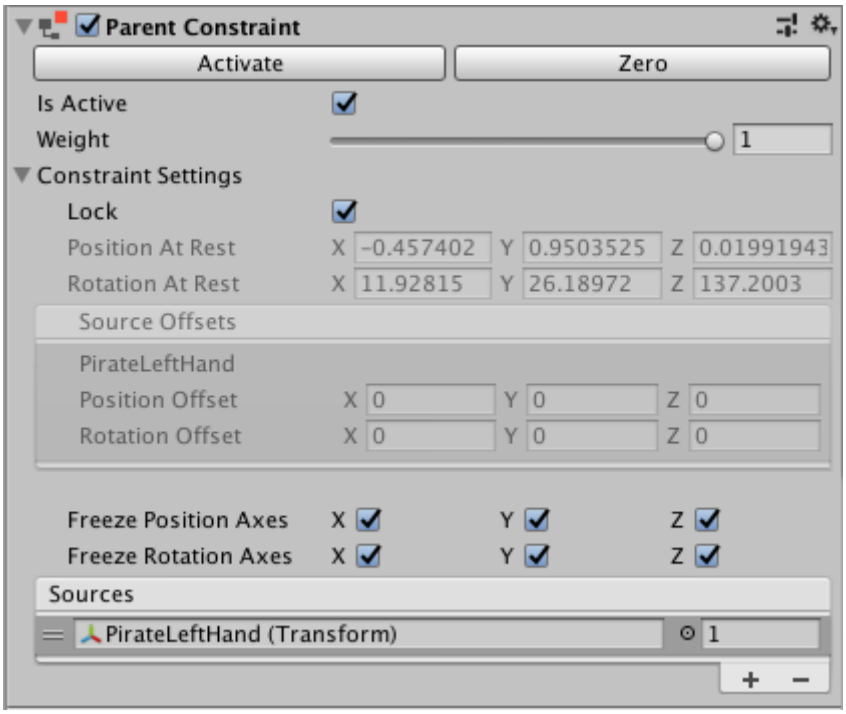
A Parent Constraint does not affect scale.

A Parent Constraint can link to multiple GameObjects.

A GameObject does not have to be a child of the GameObjects that the Parent Constraint links to.

You can vary the effect of the Constraint by specifying a weight, as well as weights for each of its source GameObjects.

For example, to place a sword in the hand of a character, add a Parent Constraint **component** to the sword GameObject. In the **Sources** list of the Parent Constraint, link to the character's hand. This way, the movement of the sword is constrained to the position and rotation of the hand.



Parent Constraint component

Properties

Property:	Function:
Activate	After you move and rotate the constrained GameObject and its source GameObjects, click **Activate** to save this information. **Activate** saves the current offset from the source GameObjects in **Rotation At Rest**, **Position At Rest**, **Position Offset**, and **Rotation Offset**, then checks **Is Active** and **Lock**.
Zero	Sets the position and rotation of the constrained GameObject to the source GameObjects. **Zero** resets the **Rotation At Rest**, **Position At Rest**, **Position Offset**, and **Rotation Offset** fields then checks **Is Active** and **Lock**.

Property:	Function:
Is Active	Toggles whether or not to evaluate the Constraint. To also apply the Constraint, make sure **Lock** is checked.
Weight	The strength of the Constraint. A weight of 1 causes the Constraint to move and rotate this GameObject at the same rate as its source GameObjects. A weight of 0 removes the effect of the Constraint completely. This weight affects all source GameObjects. Each GameObject in the **Sources** list also has a weight.
Constraint Settings	
Lock	Toggle to let the Constraint move and rotate the GameObject. Uncheck this property to edit the position and rotation of this GameObject. You can also edit the Rotation At Rest, Position At Rest, Position Offset, and Rotation Offset properties. If Is Active is checked, the Constraint updates the Rotation At Rest, Position At Rest, Position Offset, or Rotation Offset properties for you as you move and rotate the GameObject or its Source GameObjects. When you are satisfied with your changes, check Lock to let the Constraint to control this GameObject. This property has no effect in Play Mode.
Position At Rest	The X, Y, and Z values to use when Weight is 0 or when the corresponding **Freeze Position Axes** are not checked. To edit these fields, uncheck **Lock**.
Rotation At Rest	The X, Y, and Z values to use when Weight is 0 or when the corresponding **Freeze Rotation Axes** are not checked. To edit these fields, uncheck **Lock**.
Position Offset	The X, Y, and Z position offset from the Transform that the Constraint imposes. To edit these fields, uncheck **Lock**.
Rotation Offset	The X, Y, and Z rotation offset from the Transform that the Constraint imposes. To edit these fields, uncheck **Lock**.
Freeze Position Axes	Check X, Y, or Z to allow the Constraint to control the corresponding position axes. Uncheck an axis to stop the Constraint from controlling it, which allows you to edit, animate, or script it.
Freeze Rotation Axes	Check X, Y, or Z to allow the Constraint to control the corresponding rotation axes. Uncheck an axis to stop the Constraint from controlling it, which allows you to edit, animate, or script it.
Sources	The list of GameObjects that constrain this GameObject. Unity evaluates source GameObjects in the order they appear in this list. This order affects how this Constraint moves and rotates the constrained GameObject. To get the result you want, drag and drop items in this list. Each source has a weight from 0 to 1.

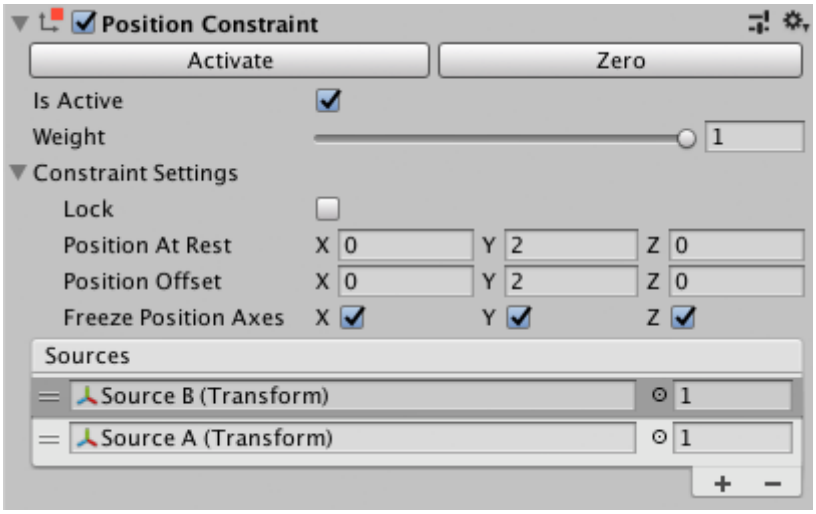
2018–03–13 Page published with editorial review

Constraints added in 2018.1

Position Constraints

Leave feedback

A Position Constraint **component** moves a GameObject to follow its source **GameObjects**.



Position Constraint component

Properties

Property:	Function:
Activate	After you position the constrained GameObject and its source GameObjects, click **Activate** to save this information. **Activate** saves the current offset from the source GameObjects in **Position At Rest** and **Position Offset**, then checks **Is Active** and **Lock**.
Zero	Sets the position of the constrained GameObject to the source GameObjects. **Zero** resets the **Position At Rest** and **Position Offset** fields, then checks **Is Active** and **Lock**.
Is Active	Toggles whether or not to evaluate the Constraint. To also apply the Constraint, make sure **Lock** is checked.
Weight	The strength of the Constraint. A weight of 1 causes the Constraint to move this GameObject at the same rate as its source GameObjects. A weight of 0 removes the effect of the Constraint completely. This weight affects all source GameObjects. Each GameObject in the **Sources** list also has a weight.
Constraint Settings	
Lock	Toggle to let the Constraint move the GameObject. Uncheck this property to edit the position of this GameObject. You can also edit the **Position At Rest** and **Position Offset** properties. If **Is Active** is checked, the Constraint updates the **At Rest** or **Offset** properties for you as you move the GameObject or its Source GameObjects. When you are satisfied with your changes, check **Lock** to let the Constraint control this GameObject. This property has no effect in Play Mode.
Position At Rest	The X, Y, and Z values to use when **Weight** is 0 or when the corresponding **Freeze Position Axes** is not checked. To edit these fields, uncheck **Lock**.
Position Offset	The X, Y, and Z offset from the Transform that is imposed by the Constraint. To edit these fields, uncheck Lock.

Property:	Function:
Freeze Position Axes	Check X, Y, or Z to allow the Constraint to control the corresponding axes. Uncheck an axis to stop the Constraint from controlling it. This allows you to edit, animate, or script the unfrozen axis.
Sources	The list of GameObjects that constrain this GameObject. Each source has a weight from 0 to 1.

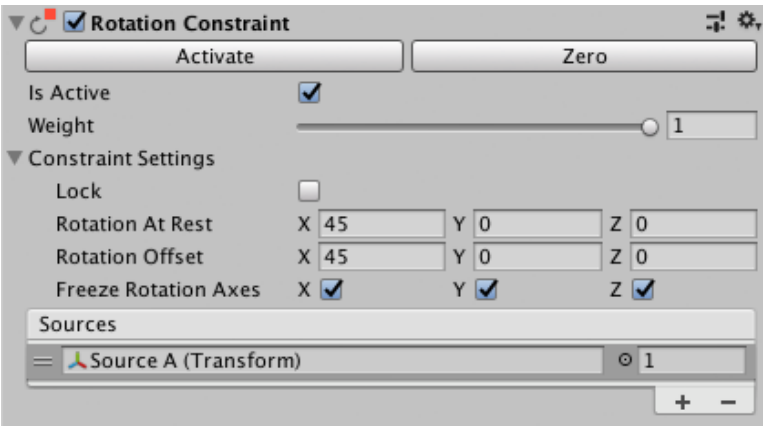
2018–03–13 Page published with editorial review

Constraints added in 2018.1

Rotation Constraints

Leave feedback

A Rotation Constraint **component** rotates a GameObject to match the rotation of its source **GameObjects**.



Rotation Constraint component

Properties

Property:	Function:
Activate	After you rotate the constrained GameObject and its source GameObjects, click **Activate** to save this information. **Activate** saves the current offset from the source GameObjects in **Rotation At Rest** and **Rotation Offset** then checks **Is Active** and **Lock**.
Zero	Sets the rotation of the constrained GameObject to the source GameObjects. **Zero** resets the **Rotation At Rest** and **Rotation Offset** fields, then checks **Is Active** and **Lock**.
Is Active	Toggles whether or not to evaluate the Constraint. To also apply the Constraint, make sure **Lock** is checked.
Weight	The strength of the Constraint. A weight of 1 causes the Constraint to rotate this GameObject at the same rate as its source GameObjects. A weight of 0 removes the effect of the Constraint completely. This weight affects all source GameObjects. Each GameObject in the **Sources** list also has a weight.
Constraint Settings	
Lock	Toggle to let the Constraint rotate the GameObject. Uncheck this property to edit the rotation of this GameObject. You can also edit the **Rotation At Rest** and **Rotation Offset** properties. If **Is Active** is checked, the Constraint updates the **Rotation At Rest** or **Rotation Offset** properties for you as you rotate the GameObject or its **Source** GameObjects. When you are satisfied with your changes, check **Lock** to let the Constraint to control this GameObject. This property has no effect in Play Mode.
Rotation At Rest	The X, Y, and Z values to use when **Weight** is 0 or when the corresponding **Freeze Rotation Axes** is not checked. To edit these fields, uncheck **Lock**.
Rotation Offset	The X, Y, and Z offset from the transform that is imposed by the Constraint. To edit these fields, uncheck **Lock**.
Freeze Rotation Axes	Check X, Y, or Z to allow the Constraint to control the corresponding axes. Uncheck an axis to stop the Constraint from controlling it. This allows you to edit, animate, or script the unfrozen axis.
Sources	The list of GameObjects that constrain this GameObject. Unity evaluates source GameObjects in the order they appear in this list. This order affects how this Constraint rotates the constrained GameObject. To get the result you want, drag and drop items in this list. Each source has a weight from 0 to 1.

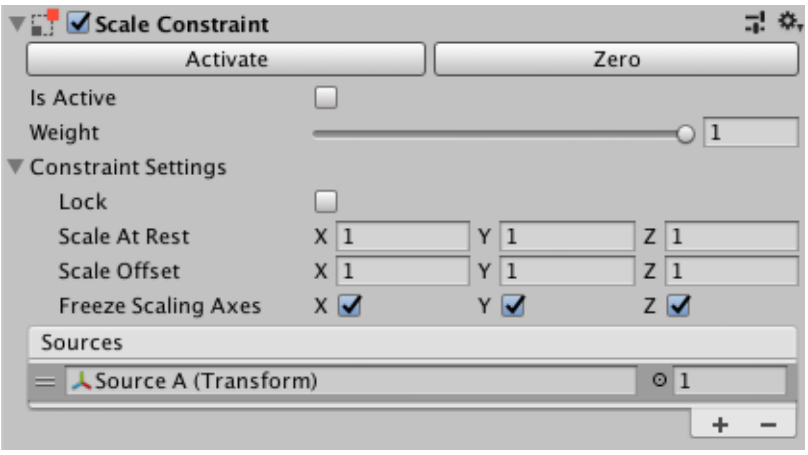
2018–04–11 Page published with editorial review

Constraints added in 2018.1

Scale Constraints

Leave feedback

A Scale Constraint **component** resizes a GameObject to match the scale of its source **GameObjects**.



Scale Constraint component

Properties

Property:	Function:
Activate	After you resize the constrained GameObject and its source GameObjects, click **Activate** to save this information. **Activate** saves the current offset from the source GameObjects in **Scale At Rest** and **Scale Offset** then checks **Is Active** and **Lock**.
Zero	Sets the rotation of the constrained GameObject to the source GameObjects. **Zero** resets the **Scale At Rest** and **Scale Offset** fields then checks **Is Active** and **Lock**.
Is Active	Toggles whether or not to evaluate the Constraint. To also apply the Constraint, make sure **Lock** is checked.
Weight	The strength of the Constraint. A weight of 1 causes the Constraint to resize this GameObject at the same rate as its source GameObjects. A weight of 0 removes the effect of the Constraint completely. This weight affects all source GameObjects. Each GameObject in the **Sources** list also has a weight.
Constraint Settings	
Lock	Toggle to let the Constraint resize the GameObject. Uncheck this property to edit the scale of this GameObject. You can also edit the **Scale At Rest** and **Scale Offset** properties. If **Is Active** is checked, the Constraint updates the **Scale At Rest** or **Scale Offset** properties for you as you resize the GameObject or its Source GameObjects. When you are satisfied with your changes, check **Lock** to let the Constraint to control this GameObject. This property has no effect in Play Mode.
Scale At Rest	The X, Y, and Z values to use when **Weight** is 0 or when the corresponding **Freeze Scale Axes** is not checked. To edit these fields, uncheck **Lock**.
Scale Offset	The X, Y, and Z offset from the transform that is imposed by the Constraint. To edit these fields, uncheck Lock.
Freeze Scale Axes	Check X, Y, or Z to allow the Constraint to control the corresponding axes. Uncheck an axis to stop the Constraint from controlling it. This allows you to edit, animate, or script the unfrozen axis.
Sources	The list of GameObjects that constrain this GameObject. Each source has a weight from 0 to 1.

2018–03–13 Page published with editorial review

Constraints added in 2018.1

Rotation and Orientation in Unity

Leave feedback

Summary

Rotations in 3D applications are usually represented in one of two ways, Quaternions or Euler angles. Each has its own uses and drawbacks. Unity uses Quaternions internally, but shows values of the equivalent Euler angles in the inspector to make it easy for you to edit.

The Difference Between Euler Angles and Quaternions

Euler Angles

Euler angles have a simpler representation, that being three angle values for X, Y and Z that are applied sequentially. To apply a Euler rotation to a particular object, each rotation value is applied in turn, as a rotation around its corresponding axis.

Benefit: Euler angles have an intuitive "human readable" format, consisting of three angles.
Benefit: Euler angles can represent the rotation from one orientation to another through a turn of more than 180 degrees
Limitation: Euler angles suffer from Gimbal Lock. When applying the three rotations in turn, it is possible for the first or second rotation to result in the third axis pointing in the same direction as one of the previous axes. This means a "degree of freedom" has been lost, because the third rotation value cannot be applied around a unique axis.

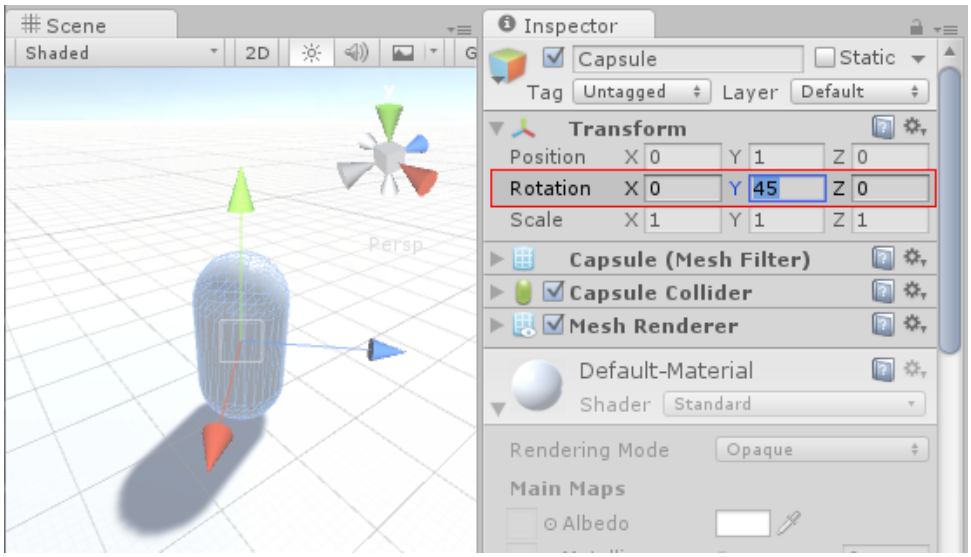
Quaternions

Quaternions can be used to represent the orientation or rotation of an object. This representation internally consists of four numbers (referenced in Unity as x, y, z & w) however these numbers don't represent angles or axes and you never normally need to access them directly. Unless you are particularly interested in delving into the mathematics of Quaternions, you only really need to know that a Quaternion represents a rotation in 3D space and you will never normally need to know or modify the x, y & z properties.

In the same way that a Vector can represent either a position or a direction (where the direction is measured from the origin), a Quaternion can represent either an orientation or a rotation - where the rotation is measured from the rotational "origin" or "Identity". It because the rotation is measured in this way - from one orientation to another - that a quaternion can't represent a rotation beyond 180 degrees.

Benefit: Quaternion rotations do not suffer from Gimbal Lock.
Limitation: A single quaternion cannot represent a rotation exceeding 180 degrees in any direction.
Limitation: The numeric representation of a Quaternion is not intuitively understandable.
In Unity all Game Object rotations are stored internally as Quaternions, because the benefits outweigh the limitations.

In the Transform Inspector however, we display the rotation using Euler angles, because this is more easily understood and edited. New values entered into the inspector for the rotation of a Game Object are converted "under the hood" into a new Quaternion rotation value for the object.



The rotation of a Game Object is displayed and edited as Euler angles in the inspector, but is stored internally as a Quaternion

As a side-effect, it is possible in the inspector to enter a value of, say, **X: 0, Y: 365, Z: 0** for a Game Object's rotation. This is a value that is not possible to represent as a quaternion, so when you hit Play you'll see that the object's rotation values change to **X: 0, Y: 5, Z: 0** (or thereabouts). This is because the rotation was converted to a Quaternion which does not have the concept of "A full 360-degree rotation plus 5 degrees", and instead has simply been set to be oriented in the same way as the result of the rotation.

Implications for Scripting

When dealing with handling rotations in your **scripts**, you should use the Quaternion class and its functions to create and modify rotational values. There are some situations where it is valid to use Euler angles, but you should bear in mind: - You should use the Quaternion Class functions that deal with Euler angles - Retrieving, modifying, and re-applying Euler values from a rotation can cause unintentional side-effects.

Creating and Manipulating Quaternions Directly

Unity's Quaternion class has a number of functions which allow you to create and manipulate rotations without needing to use Euler angles at all. For example:

Creating:

Quaternion.LookRotation
Quaternion.AngleAxis
Quaternion.FromToRotation
Manipulating:

Quaternion.Slerp
Quaternion.Inverse
Quaternion.RotateTowards
Transform.Rotate & Transform.RotateAround
However sometimes it's desirable to use Euler angles in your scripts. In this case it's important to note that you must keep your angles in variables, and only use them to *apply* them as Euler angles to your rotation. While it's possible to retrieve Euler angles *from* a quaternion, if you retrieve, modify and re-apply, problems will arise.

Here are some examples of **mistakes** commonly made using a hypothetical example of trying to rotate an object around the X axis at 10 degrees per second. This is what you should ***avoid*** :

```
// rotation scripting mistake #1
// the mistake here is that we are modifying the x value of a quaternion
```

```
// this value does not represent an angle, and will not produce desired results

void Update () {

    var rot = transform.rotation;
    rot.x += Time.deltaTime * 10;
    transform.rotation = rot;

}
```

```
// rotation scripting mistake #2
// the mistake here is that we are reading, modifying then writing the Euler
// values from a quaternion. Because these values calculated from a Quaternion,
// each new rotation may return very different Euler angles, which may suffer from gimbal

void Update () {

    var angles = transform.rotation.eulerAngles;
    angles.x += Time.deltaTime * 10;
    transform.rotation = Quaternion.Euler(angles);

}
```

And here is an example of using Euler angles in script ***correctly*** :

```
// rotation scripting with Euler angles correctly.
// here we store our Euler angle in a class variable, and only use it to
// apply it as a Euler angle, but we never rely on reading the Euler back.

float x;
void Update () {

    x += Time.deltaTime * 10;
    transform.rotation = Quaternion.Euler(x,0,0);

}
```

Implications for Animation

Many 3D authoring packages, and indeed Unity's own internal animation window, allow you to use Euler angles to specify rotations during an animation.

These rotations values can frequently exceed range expressable by quaternions. For example, if an object should rotate 720 degrees in-place, this could be represented by Euler angles X: 0, Y: 720, Z:0. But this is simply not representable by a Quaternion value.

Unity's Animation Window

Within Unity's own animation window, there are options which allow you to specify how the rotation should be interpolated - using Quaternion or Euler **interpolation**. By specifying Euler **interpolation** you are telling Unity that you want the full range of motion specified by the angles. With Quaternion rotation however, you are saying you simply want the rotation to end at a particular orientation, and Unity will use Quaternion **interpolation** and rotate across the shortest distance to get there. See Using Animation Curves for more information on this.

External Animation Sources

When importing animation from external sources, these files usually contain rotational keyframe animation in Euler format. Unity's default behaviour is to resample these animations and generate a new Quaternion keyframe for every frame in the animation, in an attempt to avoid any situations where the rotation between **keyframes** may exceed the Quaternion's valid range.

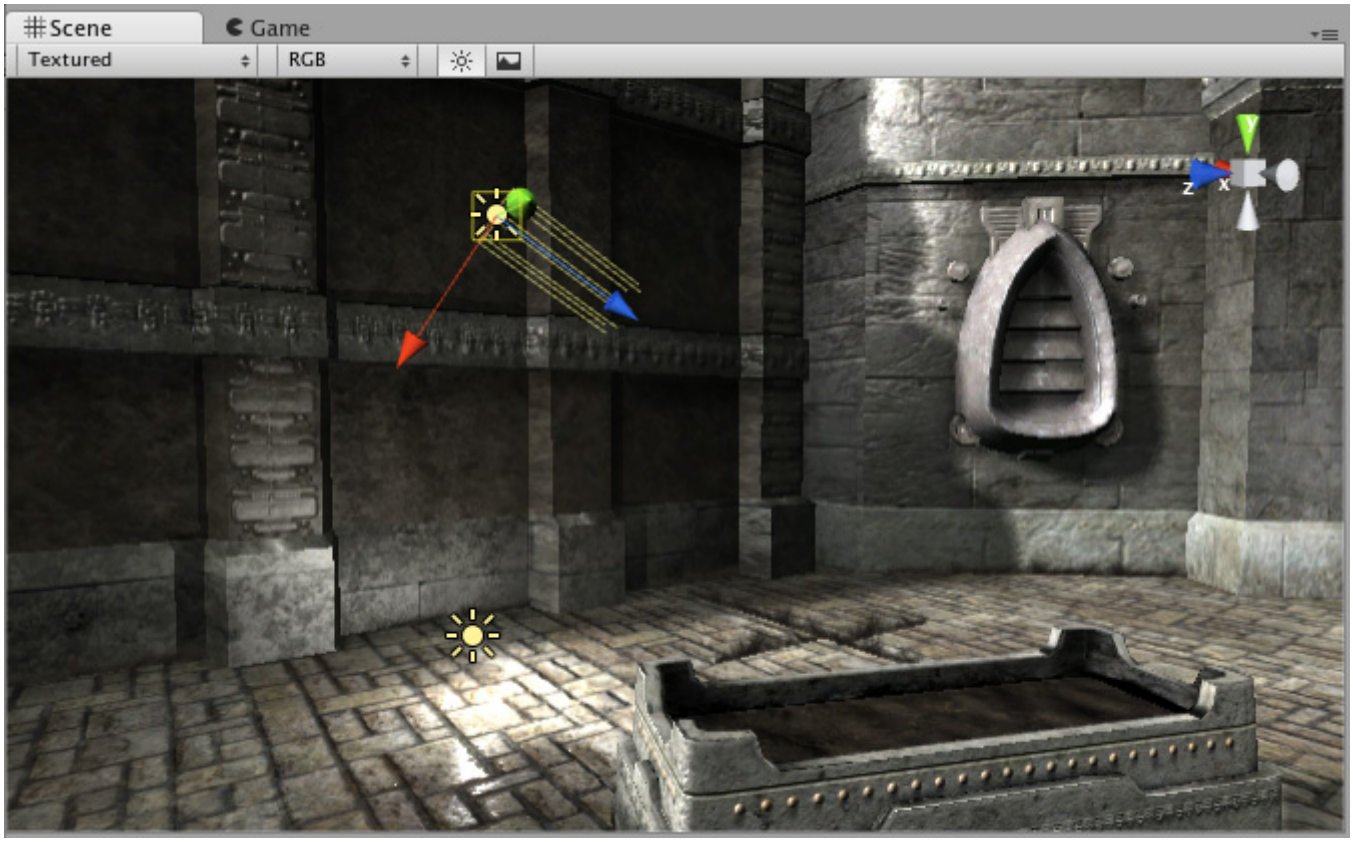
For example, imagine two keyframes, 6 frames apart, with values for X as 0 on the first keyframe and 270 on the second keyframe. Without resampling, a quaternion interpolation between these two keyframes would rotate 90 degrees in the opposite direction, because that is the shortest way to get from the first orientation to the second orientation. However by resampling and adding a keyframe on every frame, there are now only 45 degrees between keyframes so the rotation will work correctly.

There are still some situations where - even with resampling - the quaternion representation of the imported animation may not match the original closely enough, For this reason, in Unity 5.3 and onwards there is the option to turn off animation resampling, so that you can instead use the original Euler animation keyframes at runtime. For more information, see Animation Import of Euler Curve Rotations.

Lights

Leave feedback

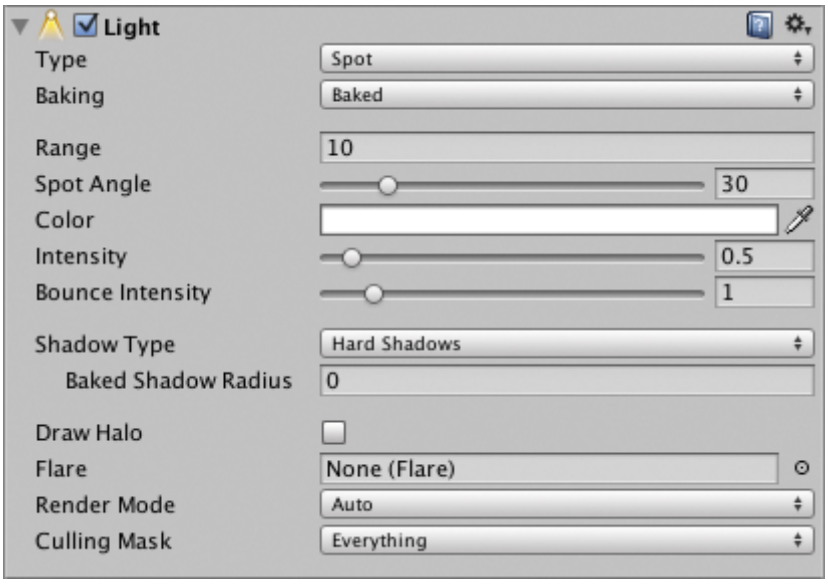
Lights are an essential part of every **scene**. While meshes and textures define the shape and look of a **scene**, lights define the color and mood of your 3D environment. You'll likely work with more than one light in each **scene**. Making them work together requires a little practice but the results can be quite amazing.



A simple, two-light setup

Lights can be added to your scene from the **GameObject->Light** menu. You will choose the light format that you want from the sub-menu that appears. Once a light has been added, you can manipulate it like any other GameObject. Additionally, you can add a Light Component to any selected GameObject by using **Component->Rendering->Light**.

There are many different options within the Light **Component** in the **Inspector**.

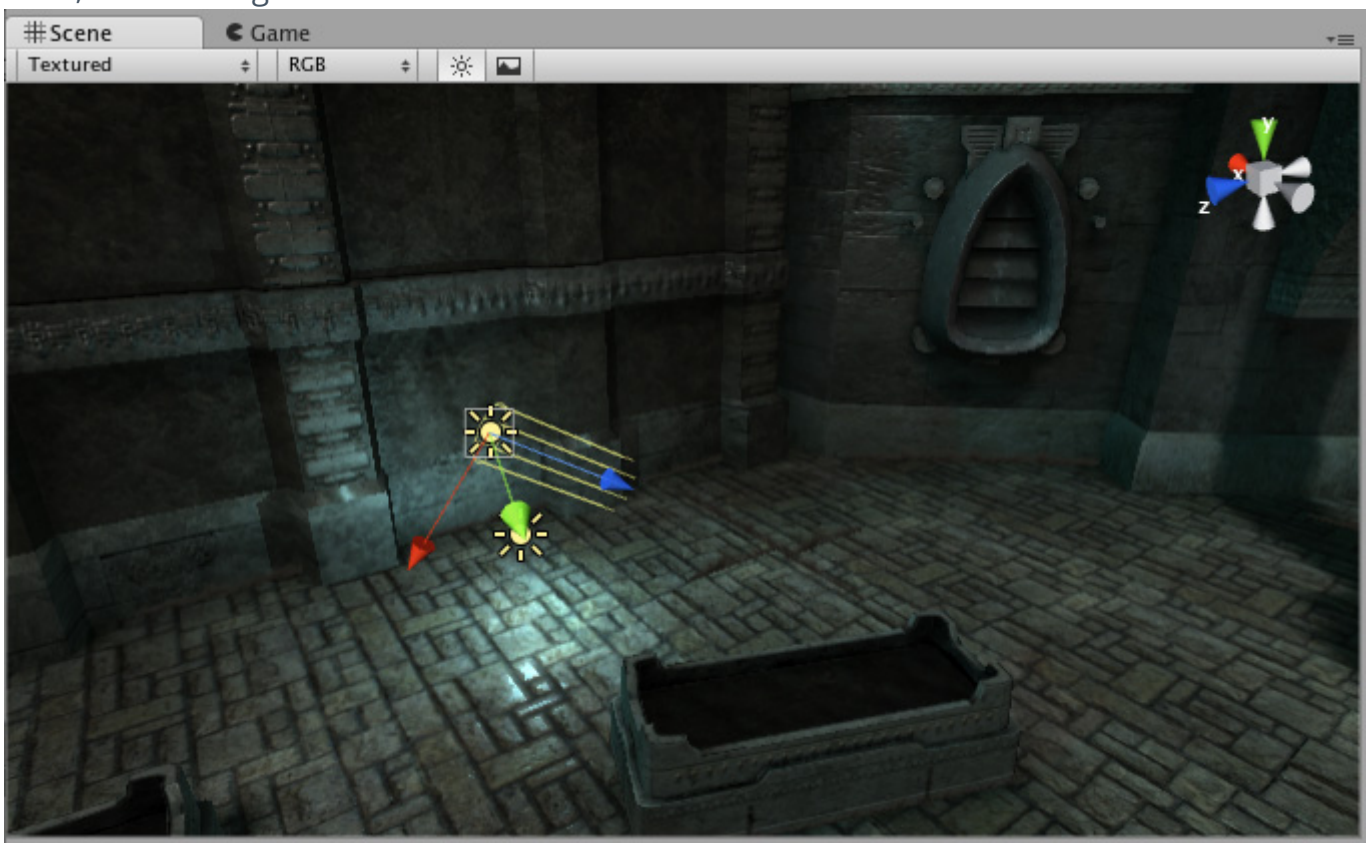


Light Component properties in the Inspector

By simply changing the **Color** of a light, you can give a whole different mood to the scene.

Bright, sunny lights

Dark, medieval lights



Spooky night lights

Rendering paths

Unity supports different **Rendering Paths**. These paths affect mainly Lights and Shadows, so choosing the correct **rendering** path depending on your game requirements can improve your project's performance. For more info

about rendering paths you can visit the Rendering paths section.

More information

For more information on how lights work see the Lighting Overview page. For more information about using the Light Component, check the Lighting Reference.

Cameras

Leave feedback

Just as cameras are used in films to display the story to the audience, **Cameras** in Unity are used to display the game world to the player. You will always have at least one camera in a **scene**, but you can have more than one. Multiple cameras can give you a two-player splitscreen or create advanced custom effects. You can animate cameras, or control them with physics. Practically anything you can imagine is possible with cameras, and you can use typical or unique cameras to fit your game's style.

For further information on how Cameras work, see the Cameras page in the Graphics Overview section.

For further information about the Camera **Component**, see the Camera Component reference page.

Adding Random Gameplay Elements

Leave feedback

Randomly chosen items or values are important in many games. This sections shows how you can use Unity's built-in random functions to implement some common game mechanics.

Choosing a Random Item from an Array

Picking an array element at random boils down to choosing a random integer between zero and the array's maximum index value (which is equal to the length of the array minus one). This is easily done using the built-in Random.Range function:-

```
var element = myArray[Random.Range(0, myArray.Length)];
```

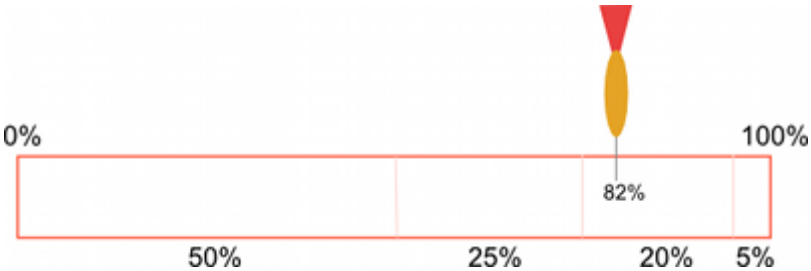
Note that Random.Range returns a value from a range that includes the first parameter but excludes the second, so using myArray.Length here gives the correct result.

Choosing Items with Different Probabilities

Sometimes, you need to choose items at random but with some items more likely to be chosen than others. For example, an NPC may react in several different ways when it encounters a player:-

50% chance of friendly greeting
25% chance of running away
20% chance of immediate attack
5% chance of offering money as a gift

You can visualise these different outcomes as a paper strip divided into sections each of which occupies a fraction of the strip's total length. The fraction occupied is equal to the probability of that outcome being chosen. Making the choice is equivalent to picking a random point along the strip's length (say by throwing a dart) and then seeing which section it is in.



In the script, the paper strip is actually an array of floats that contain the different probabilities for the items in order. The random point is obtained by multiplying Random.value by the total of all the floats in the array (they need not add up to 1; the significant thing is the relative size of the different values). To find which array element the point is "in", firstly check to see if it is less than the value in the first element. If so, then the first element is the one selected. Otherwise, subtract the first element's value from the point value and compare that to the second element and so on until the correct element is found. In code, this would look something like the following:-

```
//JS

function Choose(probs: float[]) {
    var total = 0;

    for (elem in probs) {
        total += elem;
    }

    var randomPoint = Random.value * total;

    for (i = 0; i < probs.Length; i++) {
        if (randomPoint < probs[i])
            return i;
        else
            randomPoint -= probs[i];
    }

    return probs.Length - 1;
}

//C#

    float Choose (float[] probs) {

        float total = 0;

        foreach (float elem in probs) {
            total += elem;
        }

        float randomPoint = Random.value * total;

        for (int i= 0; i < probs.Length; i++) {
            if (randomPoint < probs[i]) {
                return i;
            }
            else {
                randomPoint -= probs[i];
            }
        }
        return probs.Length - 1;
    }
```

Note that the final return statement is necessary because Random.value can return a result of 1. In this case, the search will not find the random point anywhere. Changing the line

```
if (randomPoint < probs[i])
```

...to a less-than-or-equal test would avoid the extra return statement but would also allow an item to be chosen occasionally even when its probability is zero.

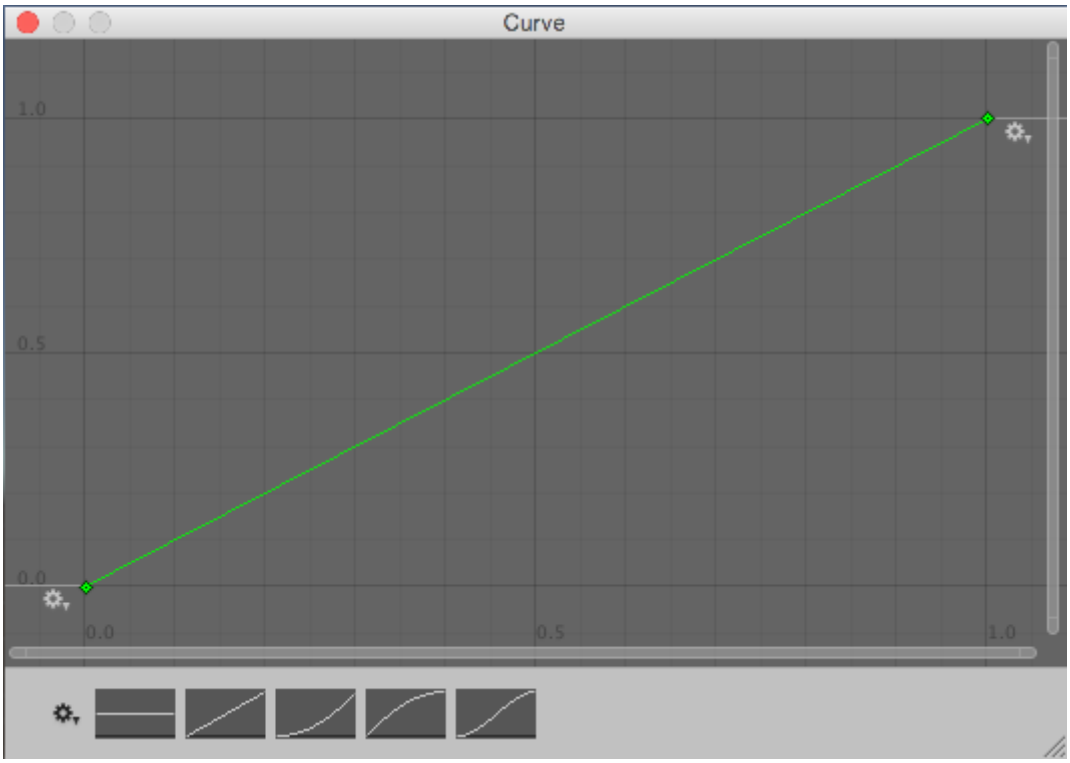
Weighting continuous random values

The array of floats method works well if you have discrete outcomes, but there are also situations where you want to produce a more continuous result - say, you want to randomize the number of gold pieces found in a treasure chest, and you want it to be possible to end up with any number between 1 and 100, but to make lower numbers more likely. Using the array-of-floats method to do this would require that you set up an array of 100 floats (i.e. sections on the paper strip) which is unwieldy; and if you aren't limited to whole numbers but instead want any number in the range, it's impossible to use that approach.

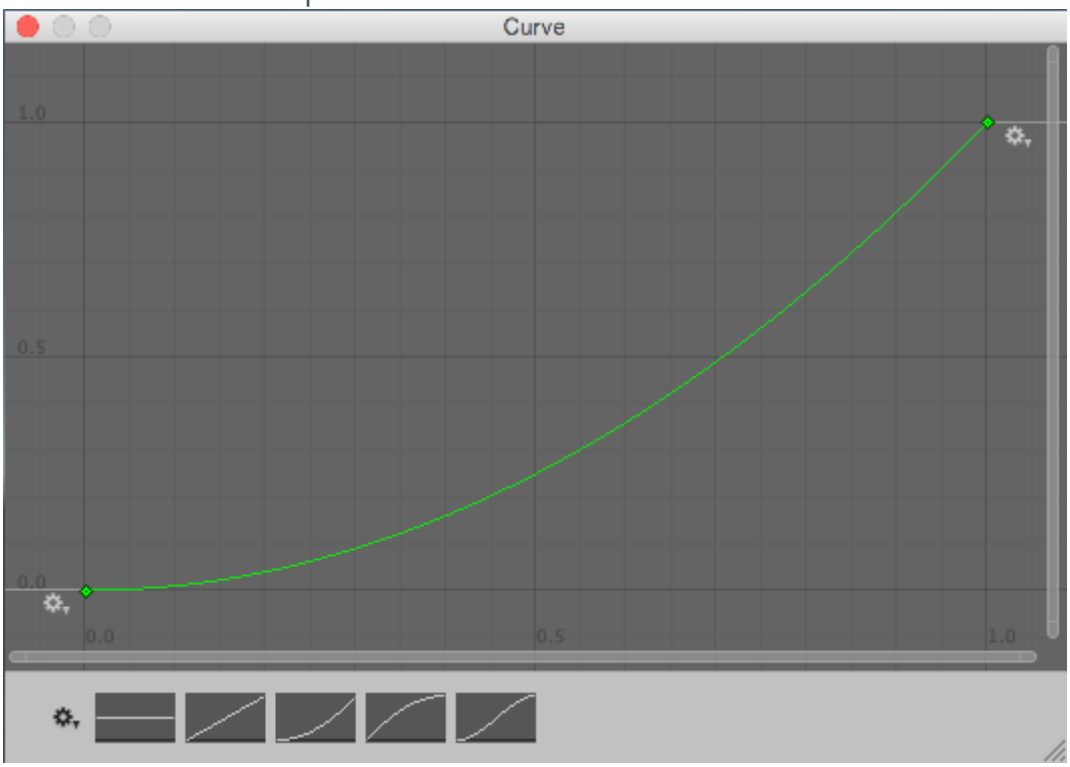
A better approach for continuous results is to use an AnimationCurve to transform a 'raw' random value into a 'weighted' one; by drawing different curve shapes, you can produce different weightings. The code is also simpler to write:

```
//JS

function CurveWeightedRandom(curve: AnimationCurve) {
    return curve.Evaluate(Random.value);
}

//C#

float CurveWeightedRandom(AnimationCurve curve) {
    return curve.Evaluate(Random.value);
}
```

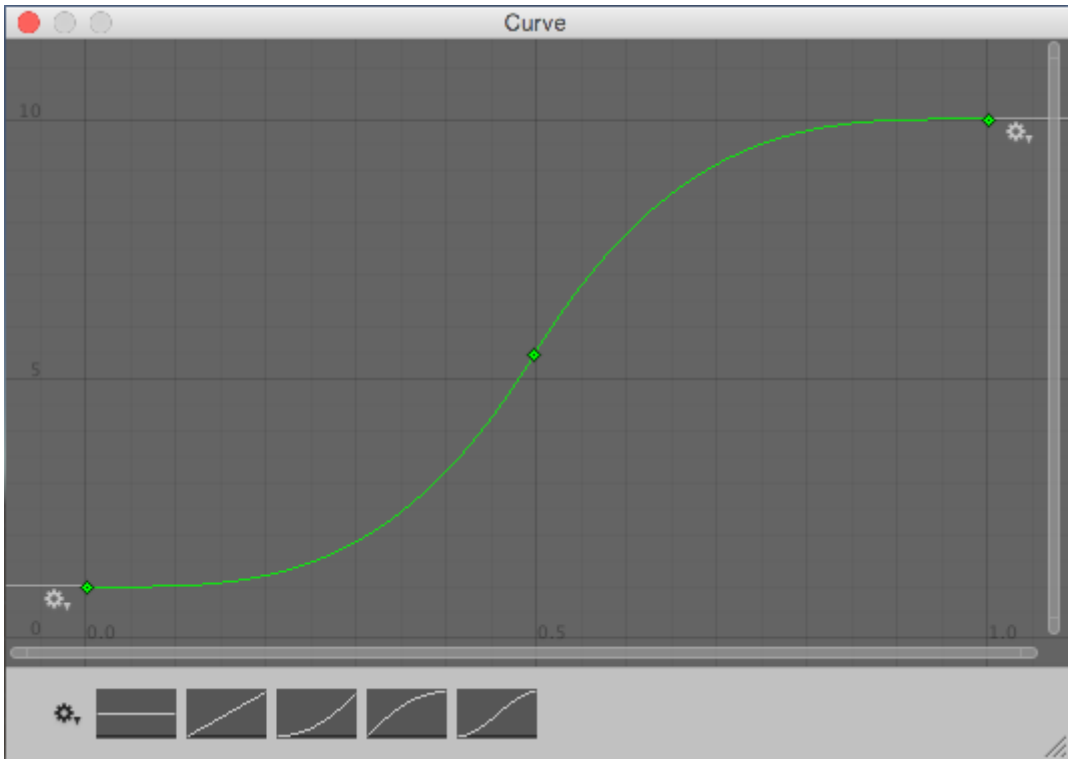
A 'raw' random value between 0 and 1 is chosen by reading from Random.value. It is then passed to curve.Evaluate(), which treats it as a horizontal coordinate, and returns the corresponding vertical coordinate of the curve at that horizontal position. Shallow parts of the curve have a greater chance of being picked, while steeper parts have a lower chance of being picked.



A linear curve does not weight values at all; the horizontal coordinate is equal to the vertical coordinate for each point on the curve.



This curve is shallower at the beginning, and then steeper at the end, so it has a greater chance of low values and a reduced chance of high values. You can see that the height of the curve on the line where x=0.5 is at about 0.25, which means there's a 50% chance of getting a value between 0 and 0.25.



This curve is shallow at both the beginning and the end, making values close to the extremes more common, and steep in the middle which will make those values rare. Notice also that with this curve, the height values have been shifted up: the bottom of the curve is at 1, and the top of the curve is at 10, which means the values produced by the curve will be in the 1–10 range, instead of 0–1 like the previous curves.

Notice that these curves are not probability distribution curves like you might find in a guide to probability theory, but are more like inverse cumulative probability curves.

By defining a public AnimationCurve variable on one of your **scripts**, you will be able to see and edit the curve through the **Inspector** window visually, instead of needing to calculate values.

This technique produces floating-point numbers. If you want to calculate an integer result - for example, you want 82 gold pieces, rather than 82.1214 gold pieces - you can just pass the calculated value to a function like Mathf.RoundToInt().

Shuffling a List

A common game mechanic is to choose from a known set of items but have them arrive in random order. For example, a deck of cards is typically shuffled so they are not drawn in a predictable sequence. You can shuffle the items in an array by visiting each element and swapping it with another element at a random index in the array:-

```
//JS

function Shuffle(deck: int[]) {
    for (i = 0; i < deck.Length; i++) {
        var temp = deck[i];
        var randomIndex = Random.Range(0, deck.Length);
        deck[i] = deck[randomIndex];
```

```
        deck[randomIndex] = temp;
    }
}

//C#

    void Shuffle (int[] deck) {
        for (int i = 0; i < deck.Length; i++) {
            int temp = deck[i];
            int randomIndex = Random.Range(0, deck.Length);
            deck[i] = deck[randomIndex];
            deck[randomIndex] = temp;
        }
    }
```

Choosing from a Set of Items Without Repetition

A common task is to pick a number of items randomly from a set without picking the same one more than once. For example, you may want to generate a number of NPCs at random spawn points but be sure that only one NPC gets generated at each point. This can be done by iterating through the items in sequence, making a random decision for each as to whether or not it gets added to the chosen set. As each item is visited, the probability of its being chosen is equal to the number of items still needed divided by the number still left to choose from.

As an example, suppose that ten spawn points are available but only five must be chosen. The probability of the first item being chosen will be 5 / 10 or 0.5. If it is chosen then the probability for the second item will be 4 / 9 or 0.44 (ie, four items still needed, nine left to choose from). However, if the first was not chosen then the probability for the second will be 5 / 9 or 0.56 (ie, five still needed, nine left to choose from). This continues until the set contains the five items required. You could accomplish this in code as follows:-

```
//JS


var spawnPoints: Transform[];

function ChooseSet(numRequired: int) {
    var result = new Transform[numRequired];

    var numToChoose = numRequired;

    for (numLeft = spawnPoints.Length; numLeft > 0; numLeft--) {
        // Adding 0.0 is simply to cast the integers to float for the division.
```

```
        var prob = (numToChoose + 0.0) / (numLeft + 0.0);

        if (Random.value <= prob) {
            numToChoose--;
            result[numToChoose] = spawnPoints[numLeft - 1];

            if (numToChoose == 0)
                break;
        }
    }

    return result;
}


//C#

    Transform[] spawnPoints;

    Transform[] ChooseSet (int numRequired) {
        Transform[] result = new Transform[numRequired];

        int numToChoose = numRequired;

        for (int numLeft = spawnPoints.Length; numLeft > 0; numLeft--) {

            float prob = (float)numToChoose/(float)numLeft;

            if (Random.value <= prob) {
                numToChoose--;
                result[numToChoose] = spawnPoints[numLeft - 1];

                if (numToChoose == 0) {
                    break;
                }
            }
        }
        return result;
    }
```

Note that although the selection is random, items in the chosen set will be in the same order they had in the original array. If the items are to be used one at a time in sequence then the ordering can make them partly predictable, so it may be necessary to shuffle the array before use.

Random Points in Space

A random point in a cubic volume can be chosen by setting each **component** of a Vector3 to a value returned by Random.value:-

```
var randVec = Vector3(Random.value, Random.value, Random.value);
```

This gives a point inside a cube with sides one unit long. The cube can be scaled simply by multiplying the X, Y and Z components of the vector by the desired side lengths. If one of the axes is set to zero, the point will always lie within a single plane. For example, picking a random point on the “ground” is usually a matter of setting the X and Z components randomly and setting the Y component to zero.

When the volume is a sphere (ie, when you want a random point within a given radius from a point of origin), you can use Random.insideUnitSphere multiplied by the desired radius:-

```
var randWithinRadius = Random.insideUnitSphere * radius;
```

Note that if you set one of the resulting vector’s components to zero, you will *not* get a correct random point within a circle. Although the point is indeed random and lies within the right radius, the probability is heavily biased toward the edge of the circle and so points will be spread very unevenly. You should use Random.insideUnitCircle for this task instead:-

```
var randWithinCircle = Random.insideUnitCircle * radius;
```

Cross-Platform Considerations

Leave feedback

Most of Unity's API and project structure is identical for all supported platforms and in some cases a project can simply be rebuilt to run on different devices. However, fundamental differences in the hardware and deployment methods mean that some parts of a project may not port between platforms without change. Below are details of some common cross-platform issues and suggestions for solving them.

Input

The most obvious example of different behaviour between platforms is in the input methods offered by the hardware.

Keyboard and joypad

The **Input.GetAxis** function is very convenient on desktop platforms as a way of consolidating keyboard and joypad input. However, this function doesn't make sense for the mobile platforms which rely on touchscreen input. Likewise, the standard desktop keyboard input doesn't port over to mobiles well for anything other than typed text. It is worthwhile to add a layer of abstraction to your input code if you are considering porting to other platforms in the future. As a simple example, if you were making a driving game then you might create your own input class and wrap the Unity API calls in your own functions:-

```
// Returns values in the range -1.0 .. +1.0 (== left .. right).
function Steering() {
    return Input.GetAxis("Horizontal");
}


// Returns values in the range -1.0 .. +1.0 (== accel .. brake).
function Acceleration() {
    return Input.GetAxis("Vertical");
}


var currentGear: int;

// Returns an integer corresponding to the selected gear.
function Gears() {
    if (Input.GetKeyDown("p"))
        currentGear++;
    else if (Input.GetKeyDown("l"))
        currentGear--;

    return currentGear;
}
```

One advantage of wrapping the API calls in a class like this is that they are all concentrated in a single source file and are consequently easy to locate and replace. However, the more important idea is that you should design your input functions according to the logical meaning of the inputs in your game. This will help to isolate the rest of the game code from the specific method of input used with a particular platform. For example, the Gears function above could be modified so that the actual input comes from touches on the screen of a mobile device. Using an integer to represent the chosen gear works fine for all platforms, but mixing the platform-specific API calls with the rest of the code would cause problems. You may find it convenient to use platform dependent compilation to combine the different implementation of the input functions in the same source file and avoid manual swaps.

Touches and clicks

The **Input.GetMouseButtonXXX** functions are designed so that they have a reasonably obvious interpretation on mobile devices even though there is no “mouse” as such. A single touch on the screen is reported as a left click and the **Input.mousePosition** property gives the position of the touch as long as the finger is touching the screen. This means that games with simple mouse interaction can often work transparently between the desktop and mobile platforms. Naturally, though, the conversion is often much less straightforward than this. A desktop game can make use of more than one mouse button and a mobile game can detect multiple touches on the screen at a time.

As with API calls, the problem can be managed partly by representing input with logical values that are then used by the rest of the game code. For example, a pinch gesture to zoom on a mobile device might be replaced by a plus/minus keystroke on the desktop; the input function could simply return a float value specifying the zoom factor. Likewise, it might be possible to use a two-finger tap on a mobile to replace a right button click on the desktop. However, if the properties of the input device are an integral part of the game then it may not be possible to remodel them on a different platform. This may mean that game cannot be ported at all or that the input and/or gameplay need to be modified extensively.

Accelerometer, compass, gyroscope and GPS

These inputs derive from the mobility of handheld devices and so may not have any meaningful equivalent on the desktop. However, some use cases simply mirror standard game controls and can be ported quite easily. For example, a driving game might implement the steering control from the tilt of a mobile device (determined by the accelerometer). In cases like this, the input API calls are usually fairly easy to replace, so the accelerometer input might be replaced by keystrokes, say. However, it may be necessary to recalibrate inputs or even vary the difficulty of the game to take account of the different input method. Tilting a device is slower and eventually more strenuous than pressing keys and may also make it harder to concentrate on the display. This may result in the game’s being more difficult to master on a mobile device and so it may be appropriate to slow down gameplay or allow more time per level. This will require the game code to be designed so that these factors can be adjusted easily.

Memory, storage and CPU performance

Mobile devices inevitably have less storage, memory and CPU power available than desktop machines and so a game may be difficult to port simply because its performance is not acceptable on lower powered hardware. Some resource issues can be managed but if you are pushing the limits of the hardware on the desktop then the game is probably not a good candidate for porting to a mobile platform.

Movie playback

Currently, mobile devices are highly reliant on hardware support for movie playback. The result is that playback options are limited and certainly don't give the flexibility that the MovieTexture **asset** offers on desktop platforms. Movies can be played back fullscreen on mobiles but there isn't any scope for using them to texture objects within the game (so it isn't possible to display a movie on a TV screen within the game, for example). In terms of portability, it is fine to use movies for introductions, cutscenes, instructions and other simple pieces of presentation. However, if movies need to be visible within the game world then you should consider whether the mobile playback options will be adequate.

Storage requirements

Video, audio and even textures can use a lot of storage space and you may need to bear this in mind if you want to port your game. Storage space (which often also corresponds to download time) is typically not an issue on desktop machines but this is not the case with mobiles. Furthermore, mobile app stores often impose a limit on the maximum size of a submitted product. It may require some planning to address these concerns during the development of your game. For example, you may need to provide cut-down versions of assets for mobiles in order to save space. Another possibility is that the game may need to be designed so that large assets can be downloaded on demand rather than being part of the initial download of the application.

Automatic memory management

The recovery of unused memory from "dead" objects is handled automatically by Unity and often happens imperceptibly on desktop machines. However, the lower memory and CPU power on mobile devices means that garbage collections can be more frequent and the time they take can impinge more heavily on performance (causing unwanted pauses in gameplay, etc). Even if the game runs in the available memory, it may still be necessary to optimise code to avoid garbage collection pauses. More information can be found on our memory management page.

CPU power

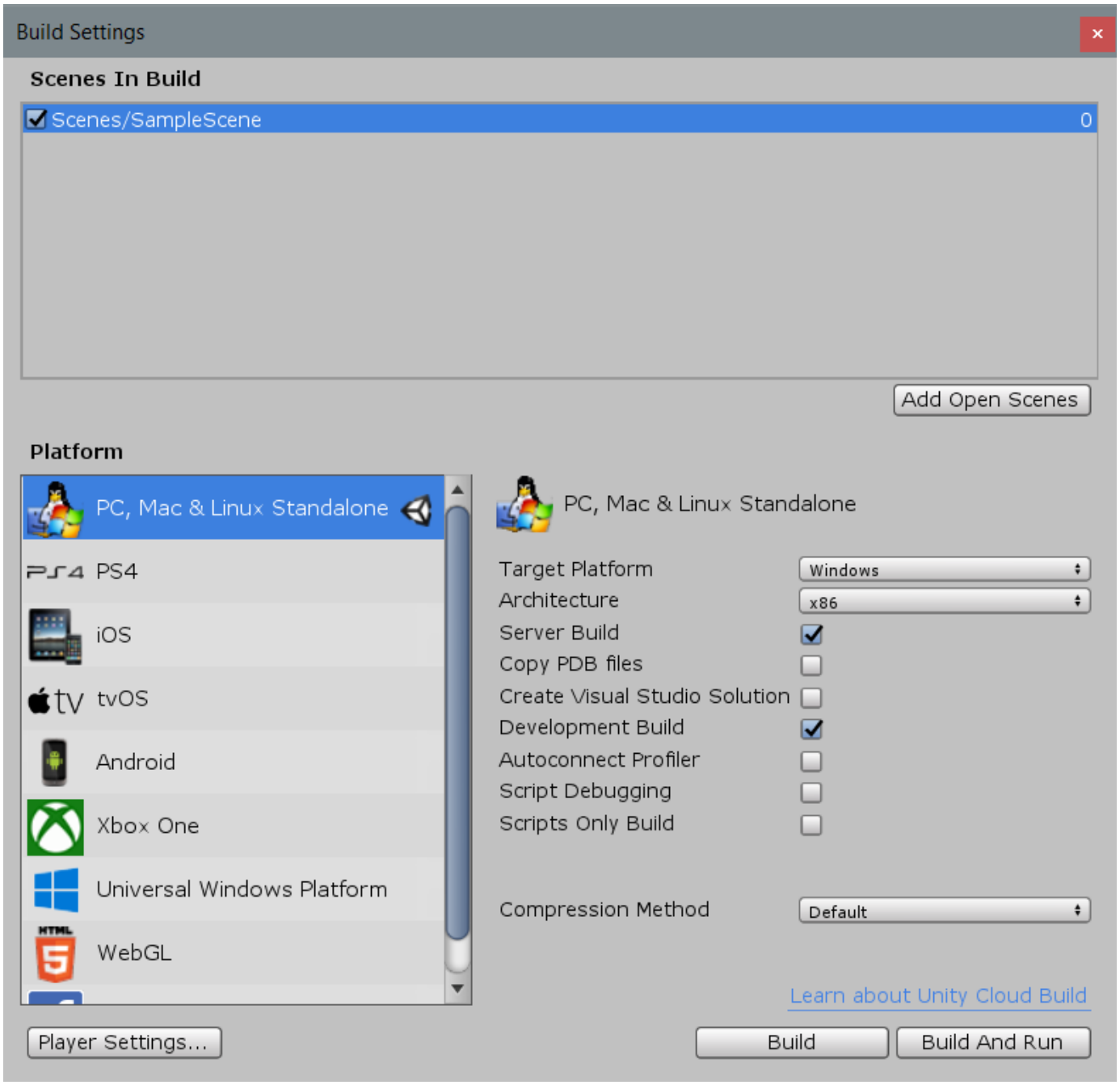
A game that runs well on a desktop machine may suffer from poor framerate on a mobile device simply because the mobile CPU struggles with the game's complexity. Extra attention may therefore need to be paid to code efficiency when a project is ported to a mobile platform. A number of simple steps to improve efficiency are outlined on this page in our manual.

Publishing Builds

Leave feedback

At any time while you are creating your game, you might want to see how it looks when you build and run it outside of the editor as a standalone. This section will explain how to access the **Build Settings** and how to create different builds of your games.

File > **Build Settings...** is the menu item to access the Build Settings window. It pops up an editable list of the scenes that will be included when you build your game.



The Build Settings window

The first time you view this window in a project, it will appear blank. If you build your game while this list is blank, only the currently open **scene** will be included in the build. If you want to quickly build a test player with only one **scene** file, just build a player with a blank **scene** list.

It is easy to add scene files to the list for multi-scene builds. There are two ways to add them. The first way is to click the **Add Open Scenes** button. You will see the currently open scenes appear in the list. The second way to add scene files is to drag them from the **Project View** to the list.

At this point, notice that each of your scenes has a different index value. **Scene 0** is the first scene that will be loaded when you build the game. When you want to load a new scene, use SceneManager.LoadScene inside your **scripts**.

If you've added more than one scene file and want to rearrange them, simply click and drag the scenes on the list above or below others until you have them in the desired order.

If you want to remove a scene from the list, click to highlight the scene and press **Command-Delete**. The scene will disappear from the list and will not be included in the build.

When you are ready to publish your build, select a **Platform** and make sure that the Unity logo is next to the platform; if its not then click in the **Switch Platform** button to let Unity know which platform you want to build for. Finally press the **Build** button. You will be able to select a name and location for the game using a standard Save dialog. When you click **Save**, Unity builds your game pronto. It's that simple. If you are unsure where to save your built game to, consider saving it into the projects root folder. You cannot save the build into the **Assets** folder.

Enabling the **Development Build** checkbox on a player will enable Profiler functionality and also make the Autoconnect Profiler and Script Debugging options available.

Further information about the Build Settings window can be found on the Build Settings page.

Building standalone players

With Unity you can build standalone applications for Windows, Mac and Linux. It's simply a matter of choosing the build target in the build settings dialog, and hitting the 'Build' button. When building standalone players, the resulting files will vary depending on the build target. For the Windows build target, an executable file (.exe) will be built, along with a Data folder which contains all the resources for your application. For the Mac build target, an app bundle will be built which contains the file needed to run the application as well as the resources.

Distributing your standalone on Mac is just to provide the app bundle (everything is packed in there). On Windows you need to provide both the .exe file and the Data folder for others to run it. Think of it like this: Other people must have the same files on their computer, as the resulting files that Unity builds for you, in order to run your game.

Inside the build process

The building process will place a blank copy of the built game application wherever you specify. Then it will work through the scene list in the build settings, open them in the editor one at a time, optimize them, and integrate them into the application package. It will also calculate all the assets that are required by the included scenes and store that data in a separate file within the application package.

Any **GameObject** in a scene that is tagged with 'EditorOnly' will not be included in the published build. This is useful for debugging scripts that don't need to be included in the final game.

When a new level loads, all the objects in the previous level are destroyed. To prevent this, use DontDestroyOnLoad() on any objects you don't want destroyed. This is most commonly used for keeping music playing while loading a level, or for **game controller** scripts which keep game state and progress.

Use SceneManager.sceneLoaded to define the message sent to all active GameObjects after the loading of a new level is finished.

For more information on creating a game with multiple Scenes, see our Tutorials.

Preloading

Published builds automatically preload all assets in a scene when the scene loads. The exception to this rule is scene 0. This is because the first scene is usually a splashscreen, which you want to display as quickly as possible.

To make sure all your content is preloaded, you can create an empty scene which calls **Application.LoadLevel(1)**. In the build settings make this empty scene's index 0. All subsequent levels will be preloaded.

You’re ready to build games

By now, you have learned how to use Unity’s interface, how to use assets, how to create scenes, and how to publish your builds. There is nothing stopping you from creating the game of your dreams. You’ll certainly learn much more along the way, and we’re here to help.

To learn more about constructing game levels, see the section on Creating Scenes.

To learn more about Scripting, see the Scripting Section.

To learn more about creating Art and importing assets, see Assets Workflow of the manual.

To interact with the community of Unity users and developers, visit the Unity Forums. You can ask questions, share projects, build a team, anything you want to do. Definitely visit the forums at least once, to showcase the games you make and find support from other developers.

Troubleshooting

Leave feedback

This section addresses common problems that can arise when using Unity. Each platform is dealt with separately below.

Platform-specific troubleshooting

Geforce 7300GT on OSX 10.6.4

Deferred **rendering** is disabled because materials are not displayed correctly for Geforce 7300GT on OX 10.6.4; This happens because of buggy video drivers.

On Windows x64, Unity crashes when script throws a NullReferenceException

You need to apply Windows hotfix #976038.

Script editing

Script opens in default system text editor, even when Visual Studio is set as the script editor

This happens when Visual Studio reports that it failed to open your script. The most common cause for this is an external **plugin** (such as Resharper) displaying a dialog at startup, requesting input from the user. This causes Visual Studio to report that it failed to open.

Graphics

Slow framerate and/or visual artifacts

This may occur if your video card drivers are not up to date. Make sure you have the latest official drivers from your card vendor.

Shadows

Shadows require certain graphics hardware support. See Light Performance page for details.
Make sure shadows are enabled in Quality Settings.
Shadows on Android and **iOS** have limitations: soft shadows are not available, and in **forward rendering** path only a single directional light can cast shadows. There is no limit to the number of lights casting shadows in the deferred **rendering path**.

Some GameObjects do not cast or receive shadows

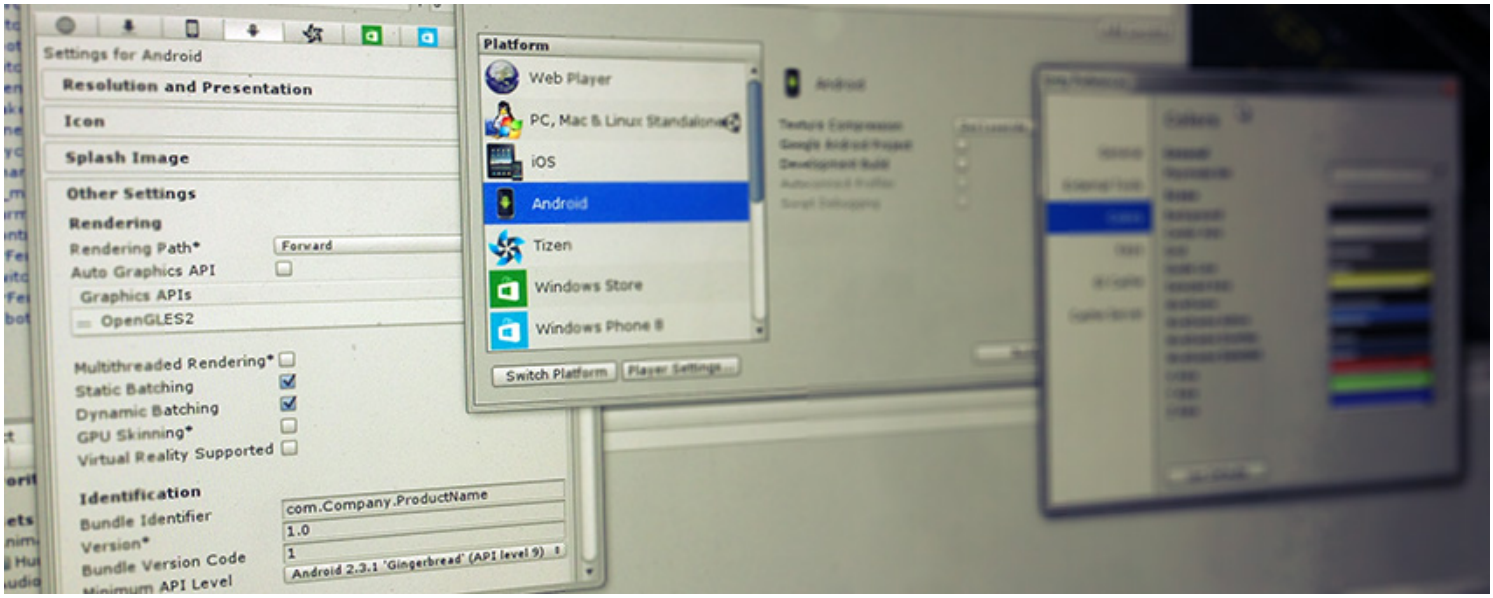
An object's Renderer must have **Receive Shadows** enabled for shadows to be rendered onto it. Also, an object must have **Cast Shadows** enabled in order to cast shadows on other objects (both are on by default).

Only opaque objects cast and receive shadows. This means that objects using the built-in Transparent or Particle shaders will not cast shadows. In most cases it is possible to use Transparent Cutout shaders for objects like fences, vegetation, **etc**. If you use custom written Shaders, they have to be pixel-lit and use the Geometry render queue. Objects using **VertexLit** shaders do not receive shadows but are able to cast them.

Only **Pixel lights** cast shadows. If you want to make sure that a light always casts shadows no matter how many other lights are in the **scene**, then you can set it to **Force Pixel** render mode (see the Light reference page).

Editor Features

Leave feedback



This section details some of the editor's basic features, which you will find useful in most projects - from choosing preferences, integrating with **version control** system, to preparing your project for a build.

2D and 3D Mode Settings

Leave feedback

When creating a new project, you can specify whether to start the Unity Editor in 2D mode or 3D mode. However, you also have the option of switching the editor between 2D Mode and 3D Mode at any time. You can read more about the difference between 2D and 3D projects here. This page provides information about how to switch modes, and what exactly changes within the editor when you do.

Switching Between 3D and 2D Modes

To change modes between 2D or 3D mode:

Bring up to the **Editor Settings Inspector**, via the **Edit>Project Settings>Editor** menu.
Then set **Default Behavior Mode** to either **2D** or **3D**.
You can find out more about the **Editor Settings Inspector** on the Editor Settings page.

Default Behavior Mode in the Editor Settings Inspector sets the project to 2D or 3D

2D vs 3D Mode Settings

2D or 3D mode determines some settings for the Unity Editor. These are listed below.

In 2D Project Mode:

Any images you import are assumed to be 2D images (**Sprites**) and set to **Sprite** mode.
The **Sprite Packer** is enabled.
The **Scene View** is set to 2D.
The default game objects do not have real time, directional light.
The **camera**'s default position is at 0,0,–10. (It is 0,1,–10 in 3D Mode.)
The camera is set to be **Orthographic**. (In 3D Mode it is **Perspective**.)
In the Lighting Window:
Skybox is disabled for new **scenes**.
Ambient Source is set to **Color**. (With the color set as a dark grey: RGB: 54, 58, 66.)
Precomputed Realtime GI is set to off.
Baked GI is set to off.
Auto-Building set to off.

In 3D Project Mode:

Any images you import are NOT assumed to be 2D images (**Sprites**).
The **Sprite Packer** is disabled.
The **Scene View** is set to 3D.
The default game objects have real time, directional light.
The camera's default position is at 0,1,–10. (It is 0,0,–10. in 2D Mode.)
The camera is set to be **Perspective**. (In 2D Mode it is **Orthographic**.)
In the Lighting Window:
Skybox is the built-in default **Skybox Material**.
Ambient Source is set to **Skybox**.

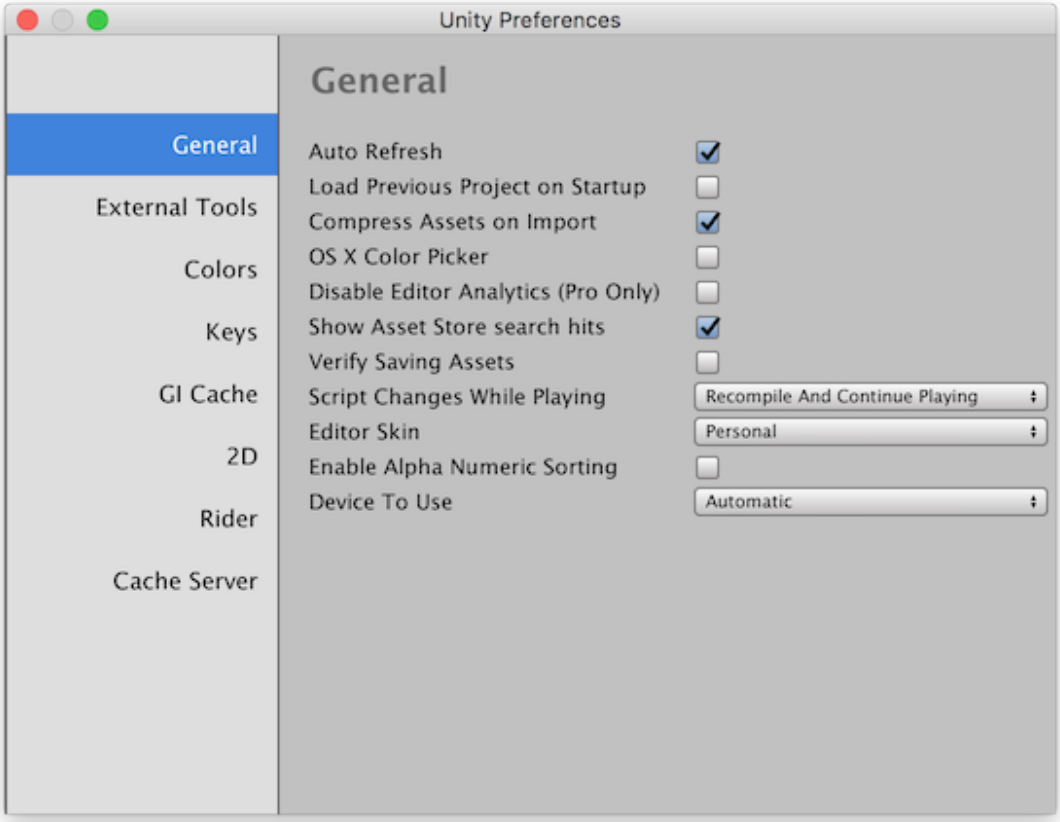
Precomputed Realtime GI is set to on.
Baked GI is set to on.
Auto-Building is set to on.

Preferences

Leave feedback

Unity provides a number of preference settings to allow you to customise the behaviour of the Unity Editor. To access them, go to **Unity** > **Preferences** (on macOS) or **Edit** > **Preferences...** (Windows).

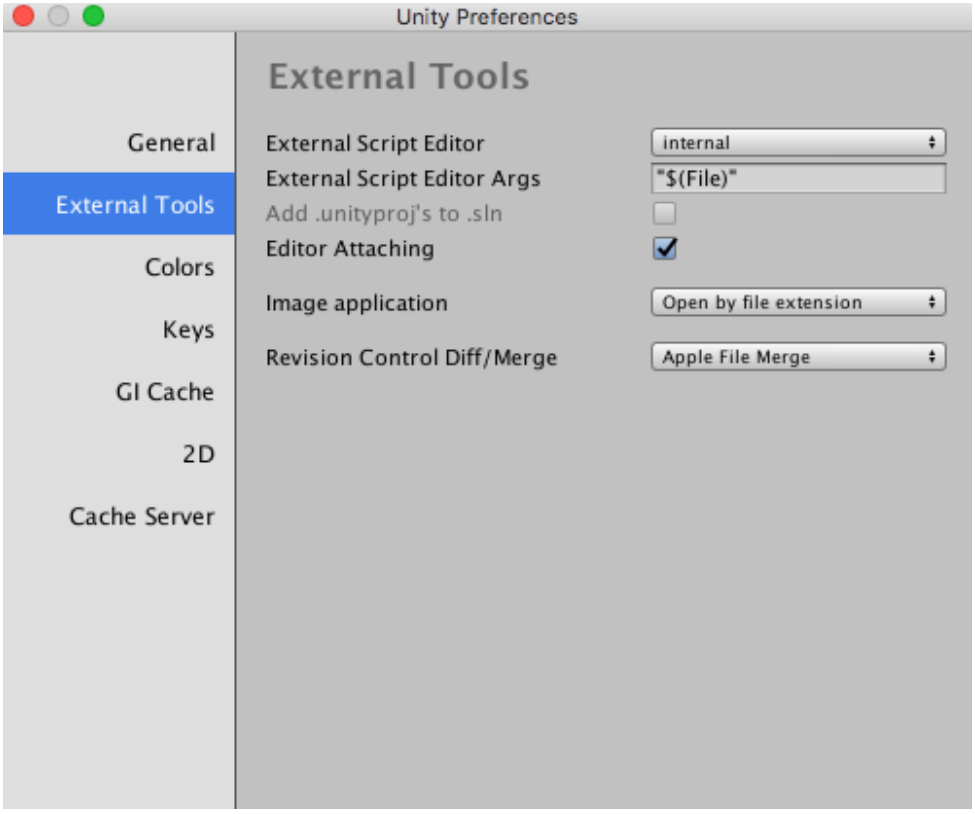
General



Setting	Properties
Auto Refresh	Check this box to update **Assets** automatically as they change.
Load Previous Project on Startup	Check this box to always load the previous project at startup.
Compress Assets on Import	Check this box to automatically compress Assets during import.
OSX Color Picker (macOS)	Check this box to use the native macOS color picker instead of Unity's own.
Disable Editor Analytics (Pro only)	Check this box to stop the Editor automatically sending information back to Unity.
Show Asset Store__A growing library of free and commercial assets created by Unity and members of the community. Offers a wide variety of assets, from textures, models and animations to whole project examples, tutorials and Editor extensions. More info See in Glossary search hits__	Check this box to show the number of free/paid Assets from the **Asset Store** in the Project Browser.

Setting	Properties
Verify Saving Assets	Check this box if you wish to verify which Assets to save individually on quitting Unity.
Script Changes While Playing	Select the drop-down menu to choose Unity's behaviour when **scripts** change while your game is running in the Editor
Recompile And Continue Playing	Recompile your scripts and keep running the **Scene**. This is the default behaviour, but you might want to change it if your scripts rely on any non-serializable data.
Recompile After Finished Playing	Defer recompilation until you manually stop your Scene, avoiding any interruption.
Stop Playing And Recompile	Immediately stop your Scene for recompilation, allowing you to quickly restart testing.
Editor Skin (Plus/Pro only)	Select the drop-down to choose which skin to apply to the Unity Editor. Choose **Personal** for light grey with black text, or **Professional** for dark grey with white text.
Enable Alpha Numeric Sorting	Check this box to enable a new button in the top-right corner of the Hierarchy window, allowing you to switch between Transform sort (which is the default behaviour) and Alphanumeric sort.
Device To Use	Select the drop-down menu to choose which of your computer's graphics devices Unity should use. You can leave this on Automatic unless you want Unity to use a specific device. This setting will override any device specified in command line options.

External tools

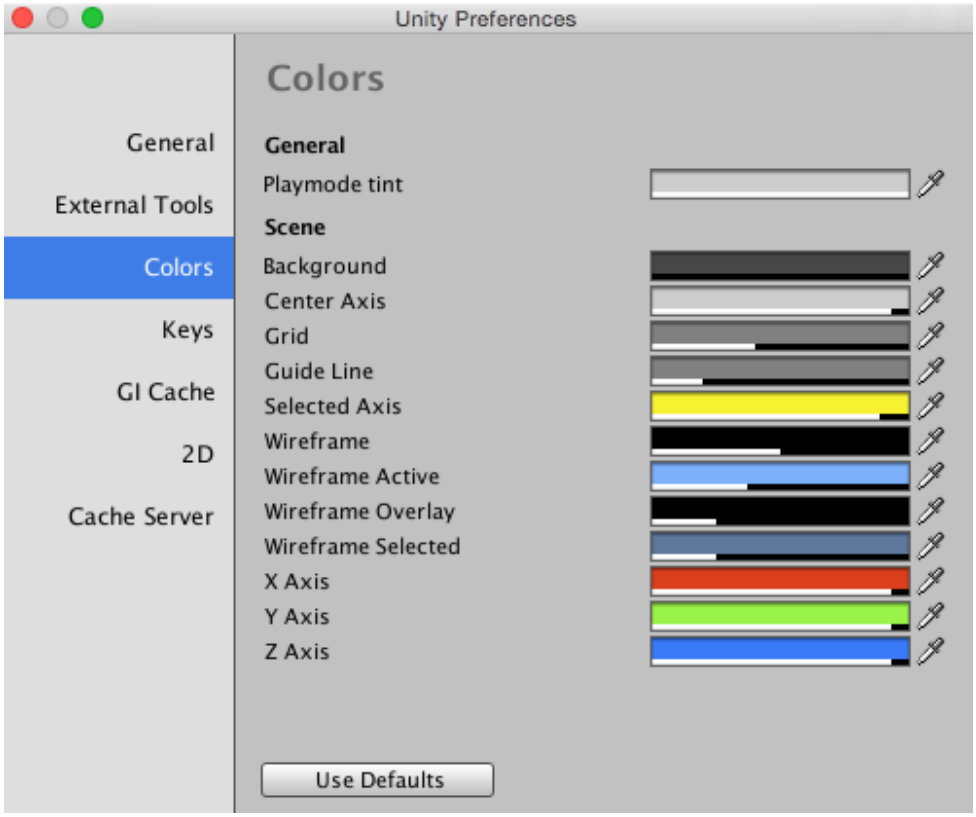


Setting	Properties
External Script Editor	Select which application Unity should use to open script files. Unity automatically passes the correct arguments to script editors it has built-in support for. Unity has built-in support for Visual Studio (Express), Visual Studio Code, Xamarin Studio and JetBrains Rider.
External Script Editor Args	Select which arguments to pass to the external script editor. `$(File)` is replaced with a path to a file being opened. `$(Line)` is replaced with a line number that editor should jump to. `$(ProjectPath)` is replaced with the path to the open project. If not set on macOS, then the default mechanism for opening files is used. Otherwise, the external script editor is only launched with the arguments without trying to open the script file using the default mechanism. See below for examples of external script editor arguments.
Editor Attaching	Check this box to allow debugging of scripts in the Unity Editor. If this option is disabled, it is not possible to attach a script debugger to Unity to debug your scripts.
Image application	Choose which application you want Unity to use to open image files.
Revision Control Diff/Merge	Choose which application you want Unity to use to resolve file differences with the **Asset server**. Unity detects these tools in their default installation locations (and checks registry keys for TortoiseMerge, WinMerge, PlasticSCM Merge, and Beyond Compare 4 on Windows).

Examples of script editor arguments

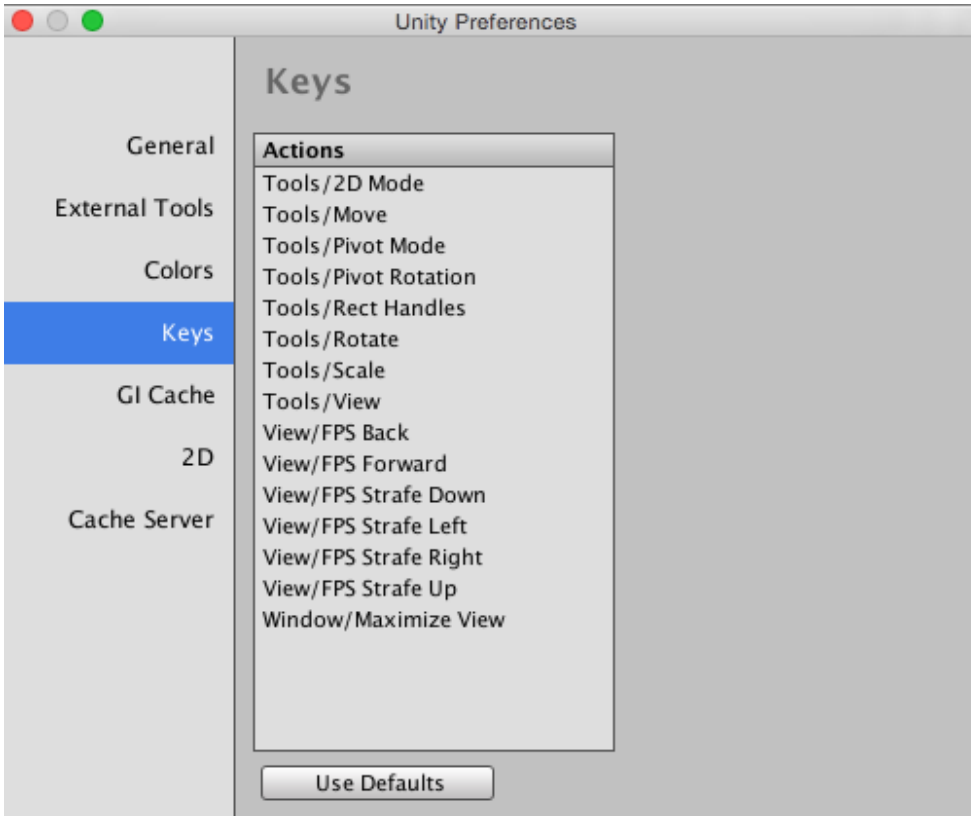
Gvim/Vim: `--remote-tab-silent +$(Line) "$File"`
Notepad2: `-g $(Line) "$(File)"`
Sublime Text 2: `"$(File)":$(Line)`
Notepad++: `-n$(Line) "$(File)"`

Colors



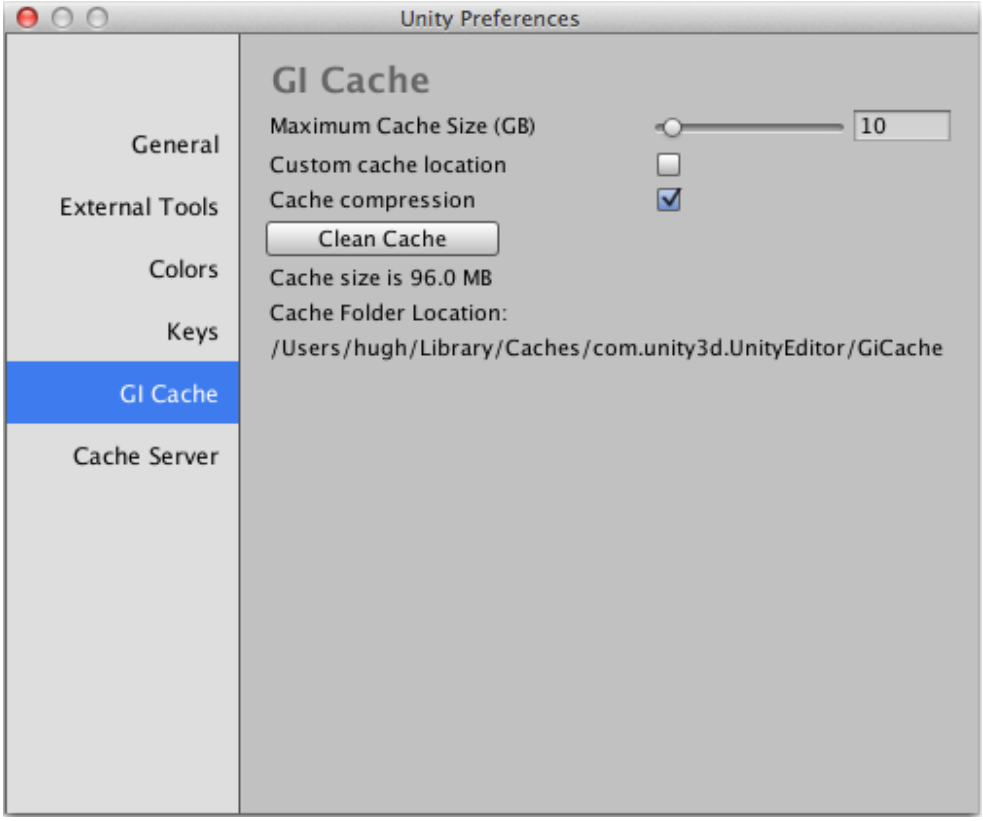
This panel allows you to choose the colors that Unity uses when displaying various user interface elements.

Keys



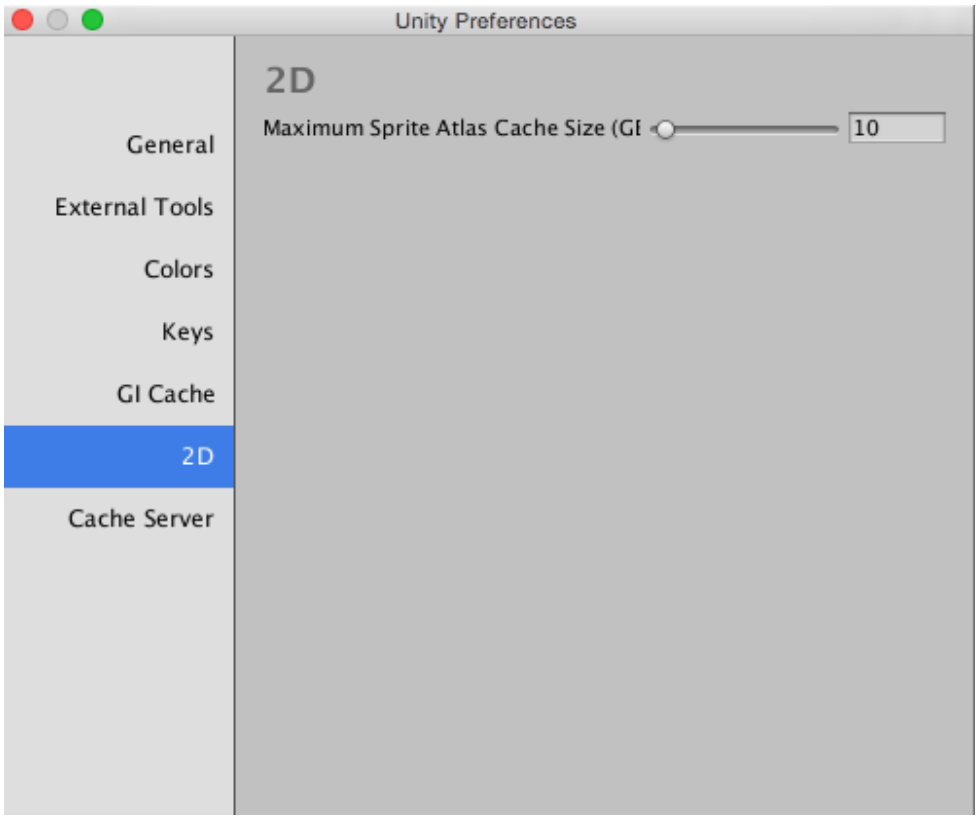
This panel allows you to set the keystrokes that activate the various commands in Unity.

GI Cache



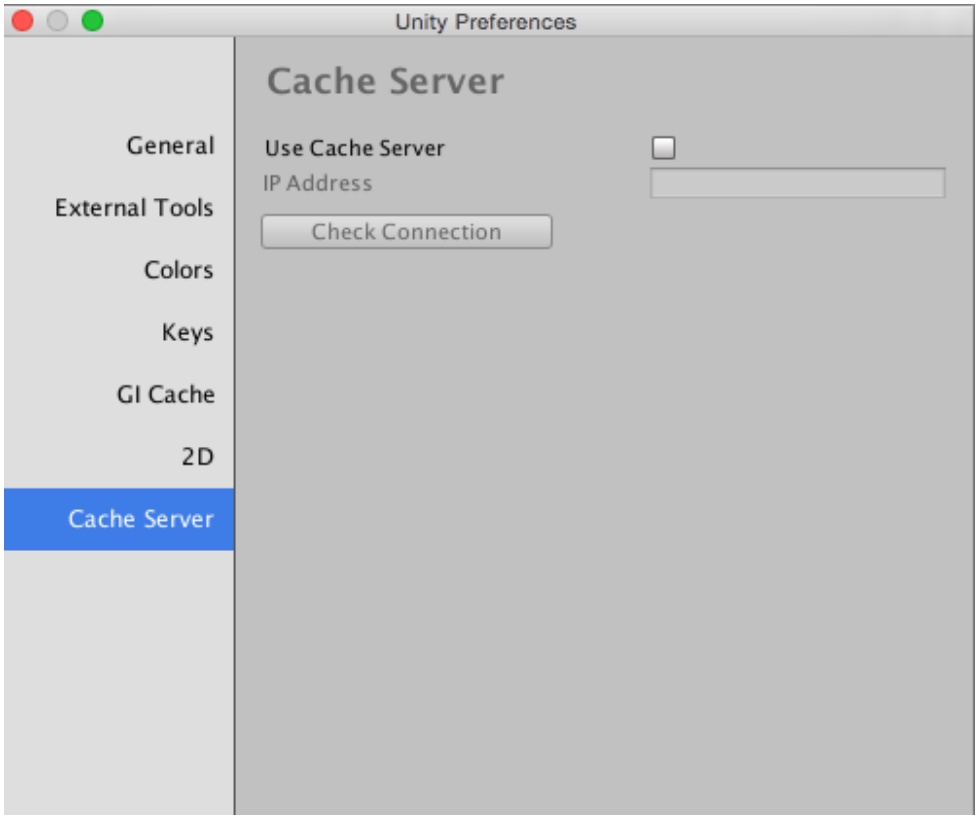
Setting	Properties
Maximum Cache Size (GB)	Use the slider to set the maximum **GI cache** folder size. The **GI cache** folder will be kept below this size whenever possible. Unused files are periodically deleted to create more space. This is carried out by the Editor automatically, and doesn't require you to do anything.
Custom cache location	Check this box to allow a custom location for the GI cache folder. The cache folder will be shared between all projects.
Cache compression	Check this box to enable a fast real-time **compression** of the GI Cache files, to reduce the size of the generated data. If you need to access the raw **Enlighten** data, disable Cache **Compression** and clean the cache.
Clean Cache	Use this button to clear the cache directory.

2D



Setting	Properties
Maximum Sprite Atlas Cache Size (GB)	Use the slider to set the maximum sprite atlas cache folder size. The sprite atlas cache folder will be kept below this size whenever possible.

Cache Server



Setting	Properties
Use Cache Server	Check this box to use a dedicated **cache server**.

Setting	Properties
IP Address	Enter the IP address of the dedicated cache server here, if enabled.

2018–04–06 Page amended with limited editorial review

Updated list of external script editors in 2018.1

Script Changes While Playing and Device To Use drop-down menus added in Unity 2018.2

Presets

Leave feedback

Use Presets to reuse property settings across multiple **components** and **assets**.

With Presets you can also specify default settings for new components and the import settings for assets. Use the **Preset Manager** to view and choose the Presets to use for default settings.

Use Presets to streamline your team's workflows. You can even use Presets to specify settings for Settings Managers, including the **Preset Manager** itself. Use this feature to configure a project then export it as a custom package. Your team members can import this package into their projects.

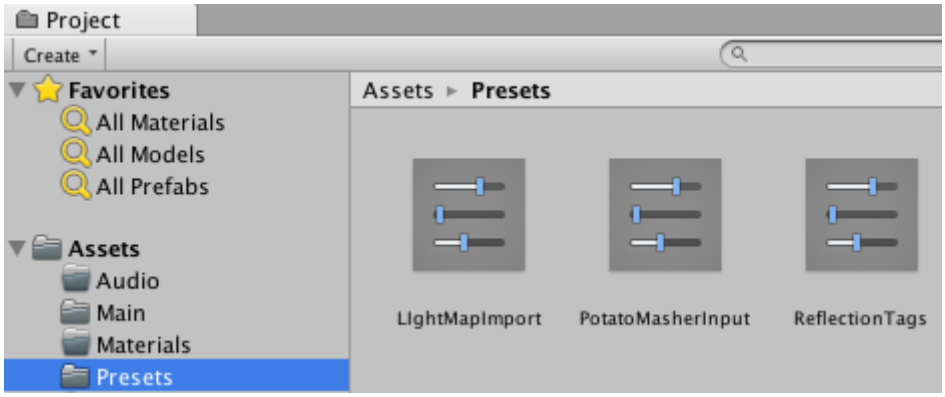
Presets are an Editor-only feature. You can support Presets in your extensions to the Unity Editor. Presets are not available at run time in the Unity Player.

Reusing property settings

You use Presets like copying and pasting. But instead of copying settings to the clipboard, you save them to use later. And like pasting settings, applying a Preset to an item changes the properties in the item.

For example, select a GameObject to edit the properties of its **RigidBody** component. Save these settings to a Preset. Then apply the Preset to **RigidBody** components in other **GameObjects**. The other components in the **GameObjects** are not affected; the Preset only applies its settings to the **RigidBody** component.

You can store Presets in the *Assets* folder of your project. Use the **Project** window to view and select Presets to edit in the **Inspector**.



Example of Preset assets in the **Project** window, organized in a *Presets* sub-folder

Saving property settings to a Preset

Use the **Select Preset** window to save property settings.

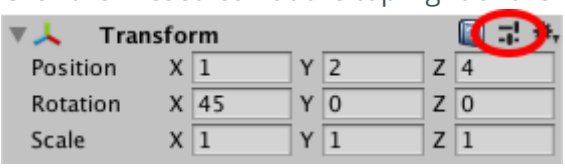
Tip: You can also save a Preset while in Play Mode.

To save settings to a Preset:

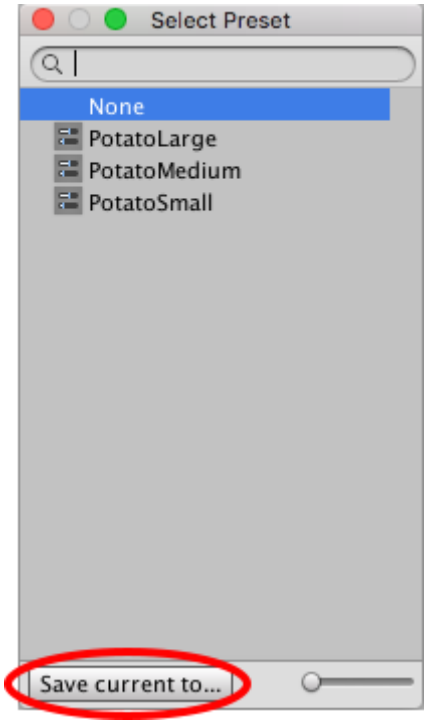
Select the GameObject, asset import settings, or Settings Manager from which you want to reuse settings.

In the **Inspector** window, edit the properties.

Click the Preset icon at the top-right of the **Inspector** window.



In the Select Preset window, click **Save current to**.



A File Save dialog appears.

Choose the location of your new Preset, enter its name, and click **Save**.

Applying a Preset

Apply a saved Preset with the **Select Preset** window or by dragging and dropping a Preset from the **Project** window onto the GameObject.

Note: Applying a Preset copies properties from the Preset to the item. It doesn't link the Preset to the item. Changes you make to the Preset do not affect the items you have previously applied the Preset to.

To apply a Preset to a Settings Manager, an existing component, or import settings for an asset:

Select the Settings Manager, GameObject, or asset import settings that you want to apply a Preset to.

In the **Inspector**, click the **Preset** icon.

In the **Select Preset** window, search for and select the Preset to apply.

Selecting the Preset applies it to the component, asset, or Settings Manager.

Close the **Select Preset** window.

Drag and drop a Preset from the **Project** window to apply properties to a component in a GameObject:

Drop the Preset on an empty spot in the **Hierarchy** window. Unity creates a new, empty GameObject and adds a component with properties copied from the Preset.

Drop the Preset on an existing GameObject in the **Hierarchy**. Unity adds a new component and copies properties from the Preset.

Drop the Preset on the **Inspector** window at the end of a GameObject. Unity adds a new component and copies properties from the Preset.

Drop the Preset on the **Inspector** onto the title of an existing component. Unity copies properties from the Preset.

Using Presets for transitions of Animation State nodes

You can save and apply Presets for Animation State nodes. However, the transitions in the Preset are shared among Presets and the nodes that you apply the Preset to. For example, you apply a Preset to two different nodes in the Animator Window. In the Inspector window, you edit the settings for one of the transitions in the first node. Your change also appears in the other node and in the Preset.

Using Presets for importing assets

You can save Presets for asset [import settings](# Presets

Use Presets to reuse property settings across multiple components and assets.

With Presets you can also specify default settings for new components and the import settings for assets. Use the **Preset Manager** to view and choose the Presets to use for default settings.

Use Presets to streamline your team's workflows. You can even use Presets to specify settings for Settings Managers, including the **Preset Manager** itself. Use this feature to configure a project then export it as a custom package. Your team members can import this package into their projects.

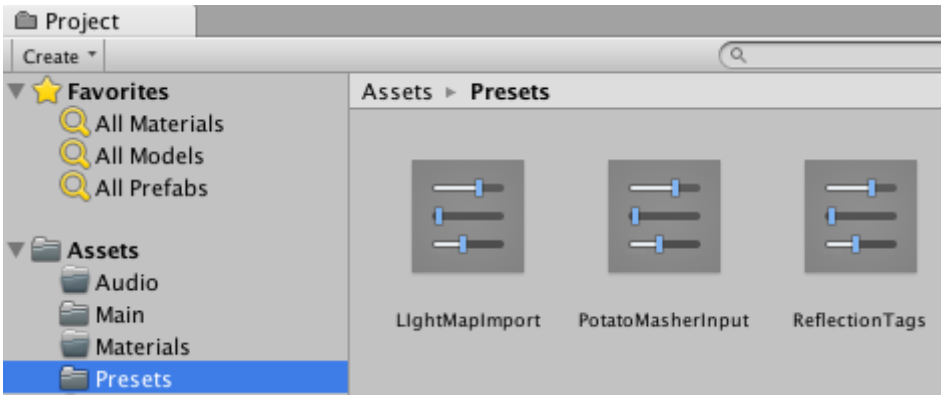
Presets are an Editor-only feature. You can support Presets in your extensions to the Unity Editor. Presets are not available at run time in the Unity Player.

Reusing property settings

You use Presets like copying and pasting. But instead of copying settings to the clipboard, you save them to use later. And like pasting settings, applying a Preset to an item changes the properties in the item.

For example, select a GameObject to edit the properties of its RigidBody component. Save these settings to a Preset. Then apply the Preset to RigidBody components in other GameObjects. The other components in the GameObjects are not affected; the Preset only applies its settings to the RigidBody component.

You can store Presets in the *Assets* folder of your project. Use the **Project** window to view and select Presets to edit in the **Inspector**.



Example of Preset assets in the **Project** window, organized in a *Presets* sub-folder

Saving property settings to a Preset

Use the **Select Preset** window to save property settings.

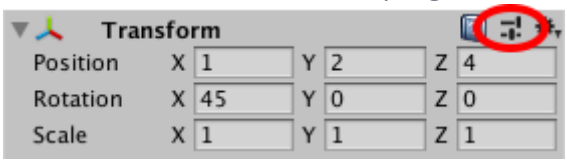
Tip: You can also save a Preset while in Play Mode.

To save settings to a Preset:

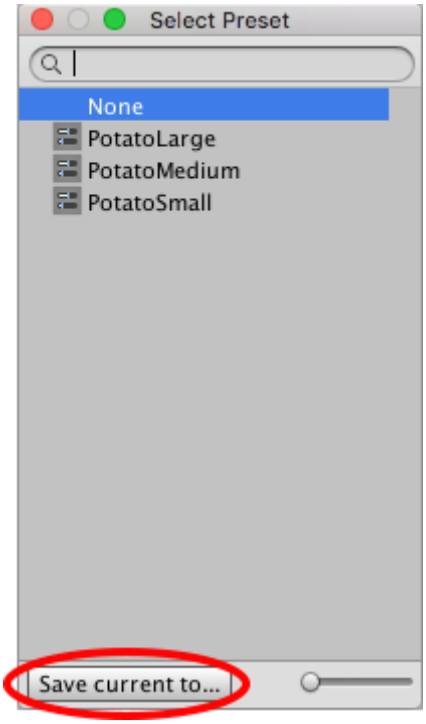
Select the GameObject, asset import settings, or Settings Manager from which you want to reuse settings.

In the **Inspector** window, edit the properties.

Click the Preset icon at the top-right of the **Inspector** window.



In the Select Preset window, click **Save current to**.



A File Save dialog appears.

Choose the location of your new Preset, enter its name, and click **Save**.

Applying a Preset

Apply a saved Preset with the **Select Preset** window or by dragging and dropping a Preset from the **Project** window onto the GameObject.

Note: Applying a Preset copies properties from the Preset to the item. It doesn't link the Preset to the item. Changes you make to the Preset do not affect the items you have previously applied the Preset to.

To apply a Preset to a Settings Manager, an existing component, or import settings for an asset:

Select the Settings Manager, GameObject, or asset import settings that you want to apply a Preset to.

In the **Inspector**, click the **Preset** icon.

In the **Select Preset** window, search for and select the Preset to apply.

Selecting the Preset applies it to the component, asset, or Settings Manager.

Close the **Select Preset** window.

Drag and drop a Preset from the **Project** window to apply properties to a component in a GameObject:

Drop the Preset on an empty spot in the **Hierarchy** window. Unity creates a new, empty GameObject and adds a component with properties copied from the Preset.

Drop the Preset on an existing GameObject in the **Hierarchy**. Unity adds a new component and copies properties from the Preset.

Drop the Preset on the **Inspector** window at the end of a GameObject. Unity adds a new component and copies properties from the Preset.

Drop the Preset on the **Inspector** onto the title of an existing component. Unity copies properties from the Preset.

Using Presets for transitions of Animation State nodes

You can save and apply Presets for Animation State nodes. However, the transitions in the Preset are shared among Presets and the nodes that you apply the Preset to. For example, you apply a Preset to two different nodes in the Animator Window. In the Inspector window, you edit the settings for one of the transitions in the first node. Your change also appears in the other node and in the Preset.

Using Presets for importing assets

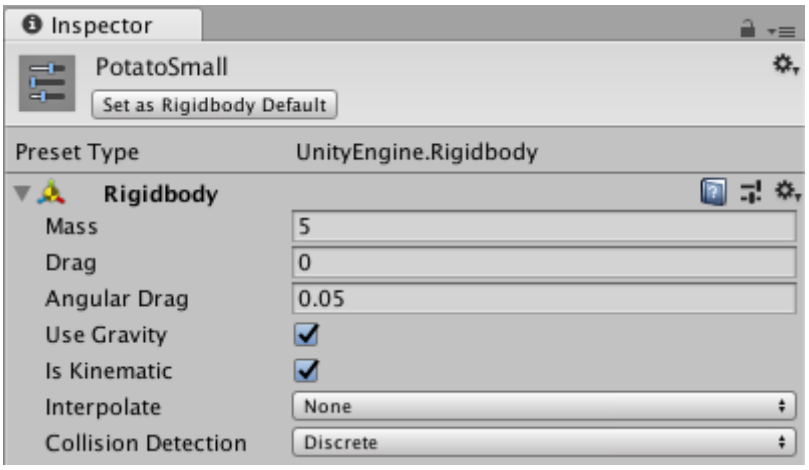
You can save Presets for asset import settings. However, applying a Preset to import settings does not affect the cross-platform settings. To apply a Preset so that it includes cross-platform settings, set the Preset as a default then use the **Reset** command.

You can also use a script to apply a Preset to an asset based on the location of the asset in the Project window.

Editing a Preset

Use the Inspector window to edit a Preset asset.

Note: Changing the properties in a Preset does not update the items that you applied the Preset to. For example, if you apply a Preset for a RigidBody component to a GameObject, then edit the Preset, the settings in the RigidBody component do not change.



Editing a Preset in the **Inspector** window

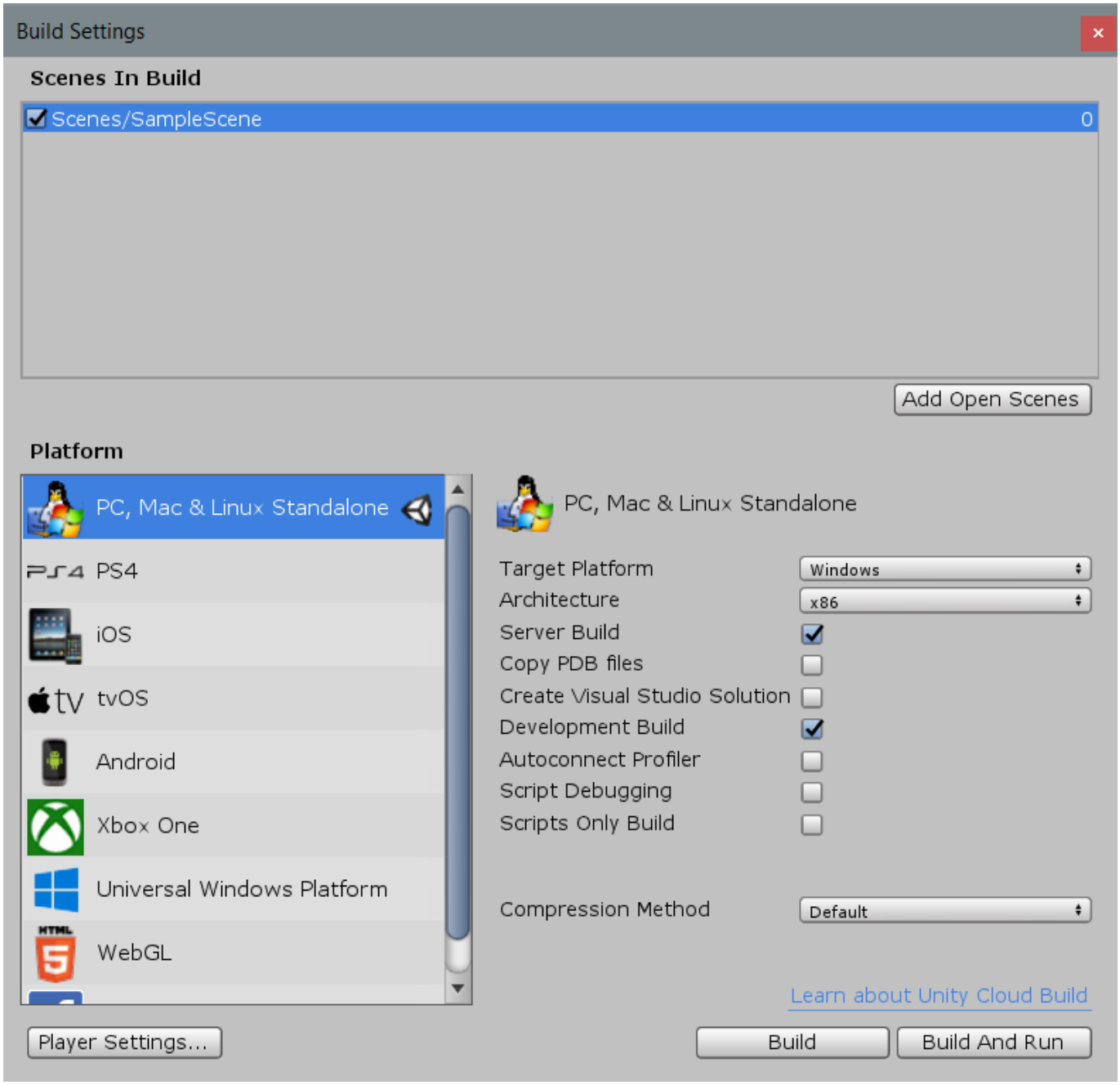
2017–03–27 Page published with limited editorial review

New feature in 2018.1

Build Settings

Leave feedback

The Build Settings window allows you to choose your target platform, adjust settings for your build, and start the build process. To access the Build Settings window, select **File** > **Build Settings...** . Once you have specified your build settings, you can click **Build** to create your build, or click the **Build And Run** to create and run your build on the platform you have specified.



Build Settings window

Scenes in Build

This part of the window shows you the scenes from your project that will be included in your build. If no scenes are shown then you can use the *Add Current* button to add the current scene to the build, or you can drag scene assets into this window from your project window. You can also untick scenes in this list to exclude them from the build without removing it from the list. If a scene is never needed in the build you can remove it from the list of scenes by pressing the delete key.

Scenes that are ticked and added to the Scenes in Build list will be included in the build. The list of scenes will be used to control the order the scenes are loaded. You can adjust the order of the scenes by dragging them up or down.

Platform List

The Platform area beneath the Scenes in Build area list all the platforms which are available to your Unity version. Some platforms may be greyed out to indicate they are not part of your version or invite you to download the platform specific build options. Selecting one of the platforms will control which platform will be built. If you change the target platform, you need to press the "Switch Platform" button to apply your change. This may take some time making the switch, because your assets may need to be re-imported in formats that match your target platform. The currently selected platform is indicated with a Unity icon to the right of the platform name.

The selected platform will show a list of options that can be adjusted for the build. Each platform may have different options. These options are listed below. Options that are common across many platforms are listed at the very bottom of this section under the "Generic items across builds" details.

PC, Mac & Linux Standalone

Option	Purpose
Target Platform	
Windows	Build for Windows
Mac OS X	Build for Mac
Linux	Build for Linux
Architecture	x86
x86	32-bit CPU
x86_64	64-bit CPU
Universal	All CPU devices
x86 + x86_64 (Universal)	All CPU devices for Linux
Server Build	Build the Player for server use and with no visual elements (headless) without the need for any command line options. When enabled, Unity builds managed **scripts** with the UNITY_SERVER define, which enables you to write server-specific code for your applications. You can also build to the Windows version as a console app so that `stdin` and `stdout` are accessible (Unity logs go to `stdout` by default).
Copy PDB files	(Windows only) Include Microsoft program database (PDB) files in the built Standalone Player. PDB files contain application debugging information that is useful for debugging, but may increase the size of your Player. This setting is disabled by default.
Headless Mode	Build game for server use and with no visual elements

iOS

Option	Purpose
Run in Xcode	Select the version of Xcode to use in the build. When set to Latest version,the build uses the most recent version of Xcode on your machine.
Run in Xcode as	
Release	Shipping version
Debug	Testing version
Symlink Unity libraries	Reference to the Unity libraries instead of copying them into the XCode project. (Reduces the XCode project size.)

Android

Option	Purpose

Option	Purpose
Texture Compression	Choose from the following texture compression formats: **Don't override** **DXT (Tegra)** **PVRTC (PowerVR)** **ATC (Adreno)** **ETC (default)** **ETC2 (GLES 3.0)** **ASTC** For advice on using these formats, see Getting started with Android development.
ETC2 fallback	For Android devices that don't support ETC2, override the default ETC2 texture decompression by choosing from 32 or 16 bit format, or 32 bit format with half the resolution.
Build System	
Internal (Default)	Generate the output package (APK) using the internal Unity build process, based on Android SDK utilities.
Gradle (New)	Generate the output package (APK) using the Gradle build system. Supports direct Build and Run and exporting the project to a directory. This is the preferred option for exporting a project, as Gradle is the native format for Android Studio.
ADT (Legacy)	Export the project in ADT (eclipse) format. The output package (APK) can be built in eclipse or Android Studio. This project type is no longer supported by Google and is considered obsolete.
SDKs for App Stores	Select which third party app stores to integrate with. To include an integration, click **Add** next to an App Store name. The Unity Package Manager automatically downloads and includes the relevant integration package.

WebGL

Build Settings for **WebGL** use the generic settings shown later on this page.

Samsung TV

Build Settings for Samsung TV use the generic settings shown later on this page.

Xiaomi

For information about building projects for Xiaomi Game Center, see Unity Xiaomi documentation.

Other platforms

Console platforms and devices which require a Unity license will be documented in the Platform Specific section of the User Guide.

Generic items across builds

Option	Purpose
Development Build	Allow the developer to test and work out how the build is coming along.
Autoconnect **Profiler**	When the Development Build option is selected allow the **profiler** to connect to the build.
Script Debugging	When the Development Build option is selected allow the script code to be debugged. Not available on WebGL.
Scripts Only Build	Build just the scripts in the current Project.
Compression Method	Compress the data in your Project when building the Player. This includes Assets, Scenes, Player Settings and GI data. Choose between the following methods:
Default	On PC, Mac, Linux Standalone, and **iOS**, there is no compression by default. On Android, the default compression is ZIP, which gives slightly better compression results than LZ4HC, but data is slower to decompress.

Option	Purpose
LZ4	A fast compression format that is useful for development builds. For more information, see BuildOptions.CompressWithLz4.
LZ4HC	A high compression variant of LZ4 that is slower to build but produces better results for release builds. For more information, see BuildOptions.CompressWithLz4HC.

2018–09–18 Page amended with limited editorial review

Xiaomi build target added in 2017.2

Tizen support discontinued in 2017.3

Server Build option added for Standalone Player Build Settings in Unity 2018.3

Project Settings

Leave feedback

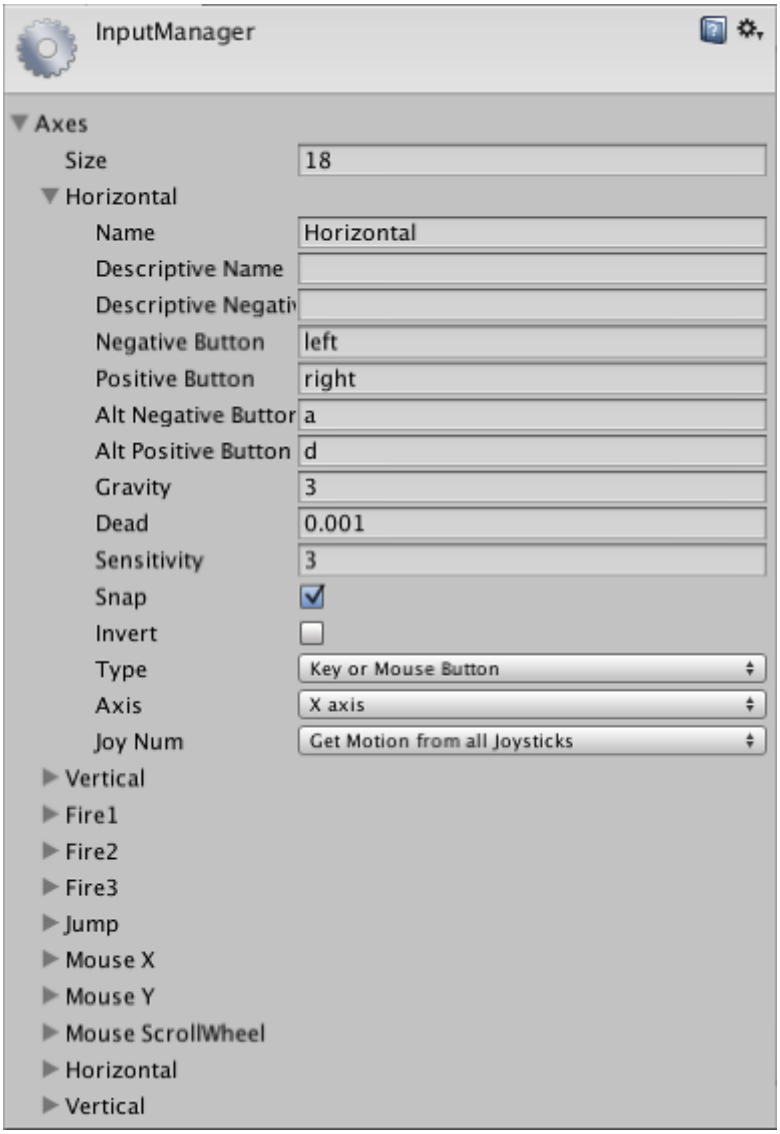
Project Settings are available from the menu **Edit > Project Settings**. Use these settings to adjust the graphics, physics, and other details of the published player. This section describes the various settings.

2018–04–23 Page amended with editorial review

Input Manager

Leave feedback

The **Input Manager** is where you define all the different input axes and game actions for your project.



The Input Manager

To see the Input Manager choose: **Edit** > **Project Settings** > **Input**. Note that the Input clsss provides more details.

Properties

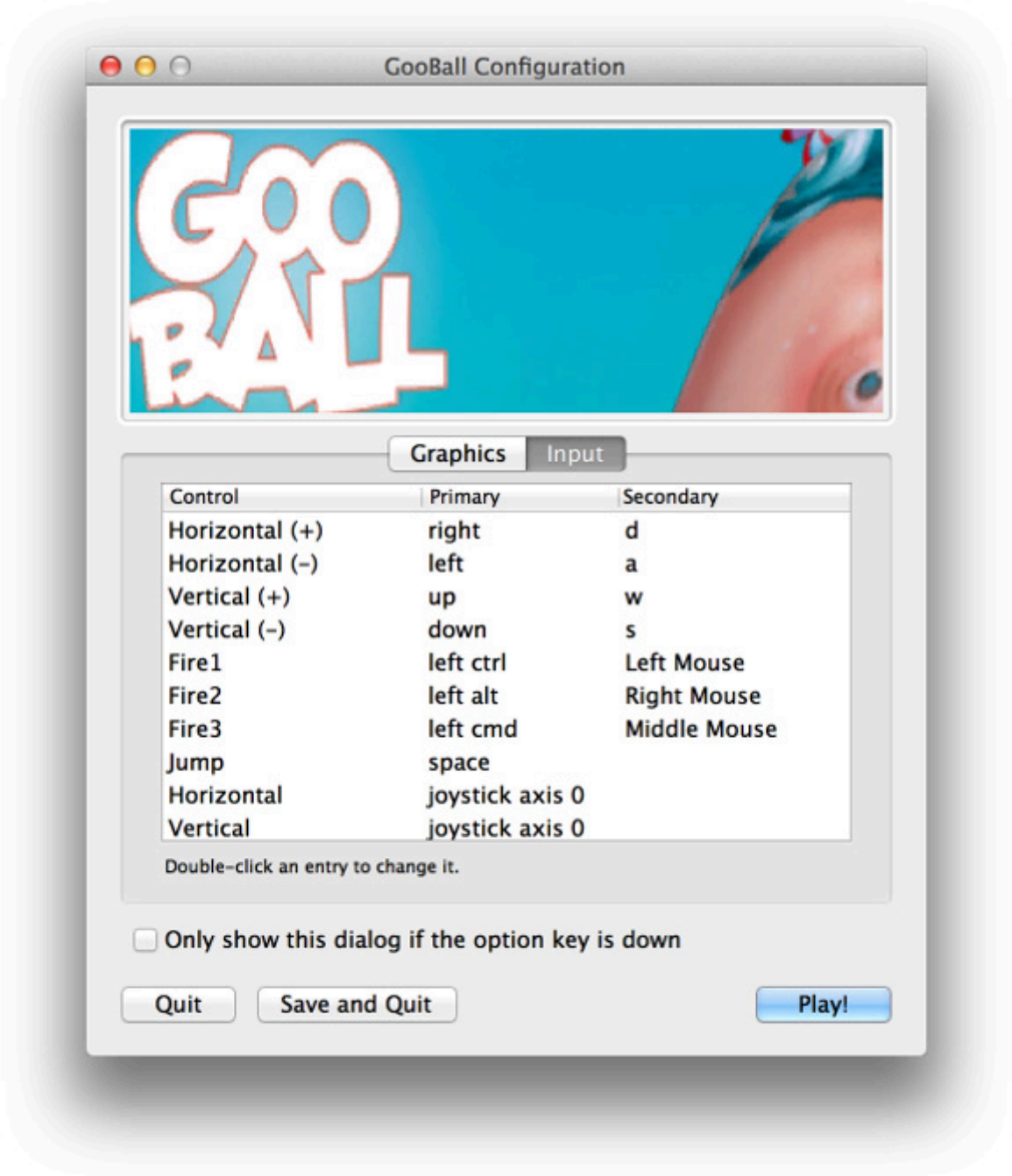
Property:	*Function:*
Axes	Contains all the defined input axes for the current project: **Size** is the number of different input axes in this project, **Element 0, 1, ...** are the particular axes to modify.
Name	The string that refers to the axis in the game launcher and through scripting.
Descriptive Name	A detailed definition of the **Positive Button** function that is displayed in the game launcher.
Descriptive Negative Name	A detailed definition of the **Negative Button** function that is displayed in the game launcher.

Property:	*Function:*
Negative Button	The button that will send a negative value to the axis.
Positive Button	The button that will send a positive value to the axis.
Alt Negative Button	The secondary button that will send a negative value to the axis.
Alt Positive Button	The secondary button that will send a positive value to the axis.
Gravity	How fast will the input recenter. Only used when the **Type** is **key / mouse button**.
Dead	Any positive or negative values that are less than this number will register as zero. Useful for joysticks.
Sensitivity	For keyboard input, a larger value will result in faster response time. A lower value will be more smooth. For Mouse delta the value will scale the actual mouse delta.
Snap	If enabled, the axis value will be immediately reset to zero after it receives opposite inputs. Only used when the **Type** is **key / mouse button**.
Invert	If enabled, the positive buttons will send negative values to the axis, and vice versa.
Type	Use **Key / Mouse Button** for any kind of buttons, Mouse Movement for mouse delta and scrollwheels, **Joystick Axis** for analog joystick axes and **Window Movement** for when the user shakes the window.
Axis	Axis of input from the device (joystick, mouse, gamepad, etc.)
Joy Num	Which joystick should be used. By default this is set to retrieve the input from all joysticks. This is only used for input axes and not buttons.

Details

All the axes that you set up in the Input Manager serve two purposes:

They allow you to reference your inputs by axis name in scripting
They allow the players of your game to customize the controls to their liking
All defined axes will be presented to the player in the game launcher, where they will see its name, detailed description, and default buttons. From here, they will have the option to change any of the buttons defined in the axes. Therefore, it is best to write your **scripts** making use of axes instead of individual buttons, as the player may want to customize the buttons for your game.



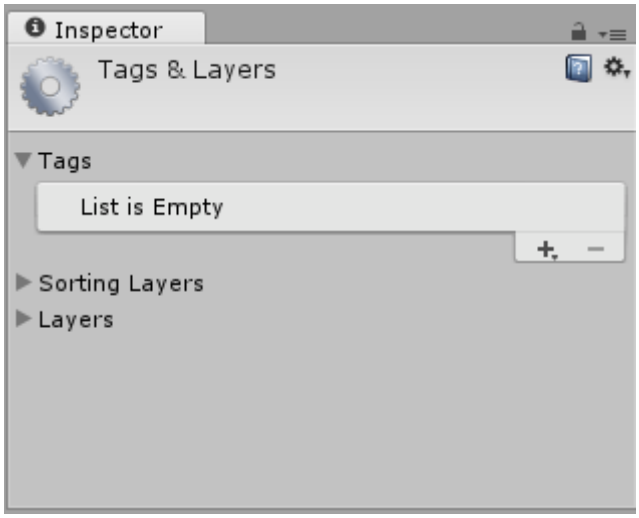
The game launcher's Input window is displayed when your game is run

See also: .

Tags and Layers

Leave feedback

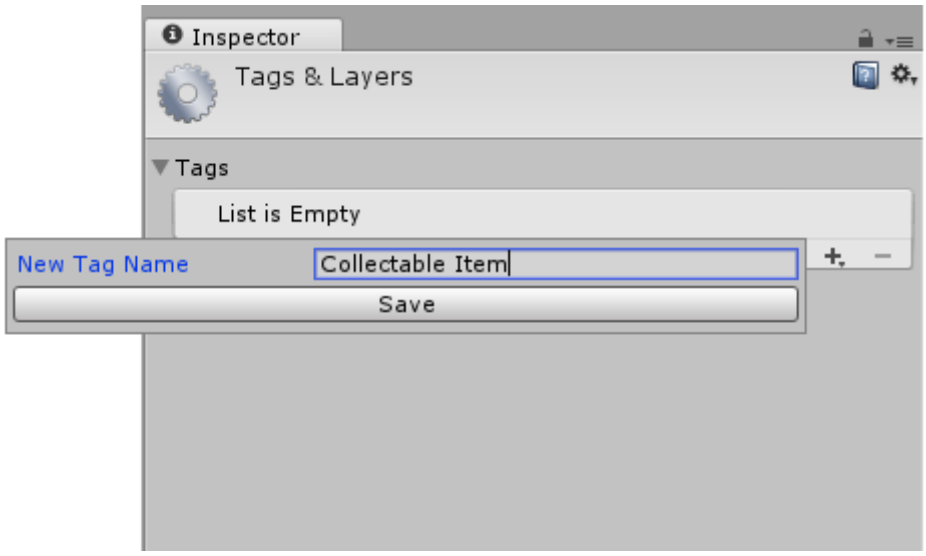
The **Tags and Layers Manager** allows you to set up **Tags**, **Sorting Layers** and **Layers**. To view the Tags and Layers Manager, go to **Edit** > **Project Settings** > **Tags and Layers**.



The Tags and Layers Manager, before any custom tags or layers have been defined

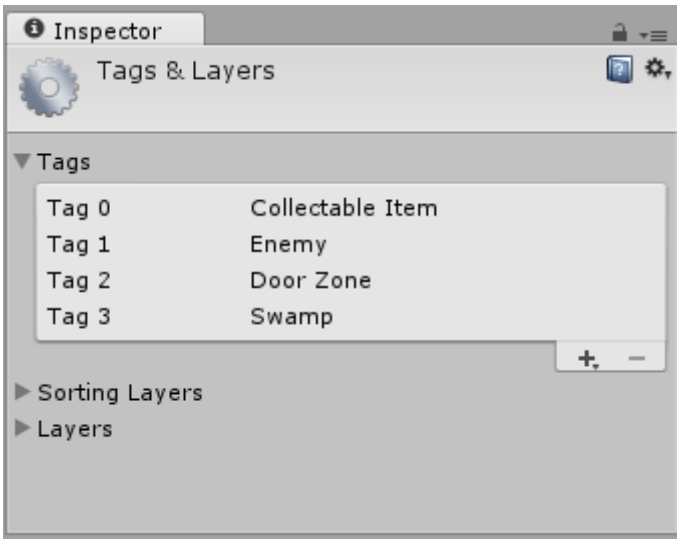
Details

Tags: These are marker values that that you can use to identify objects in your project (see documentation on Tags for further details). To add a new Tag, click the plus button (+) at the bottom-right of the list, and name your new Tag.



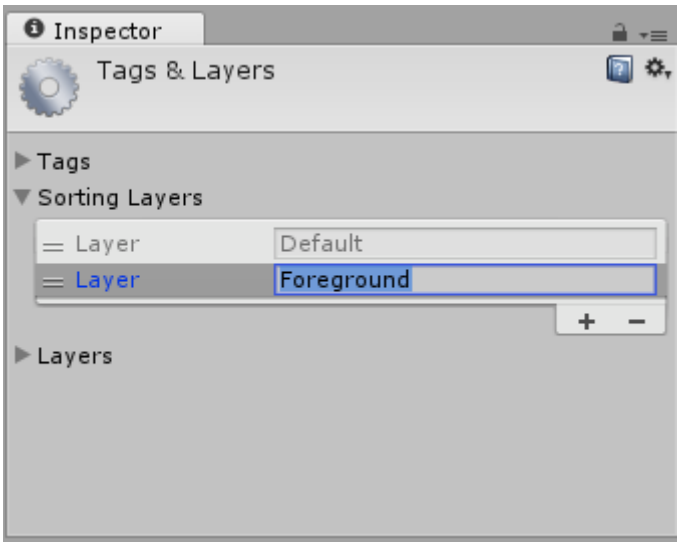
Adding a new Tag

Note that once you have named a Tag, you cannot rename it. To remove a Tag, click on it and then click the minus (-) button at the bottom-right of the list.



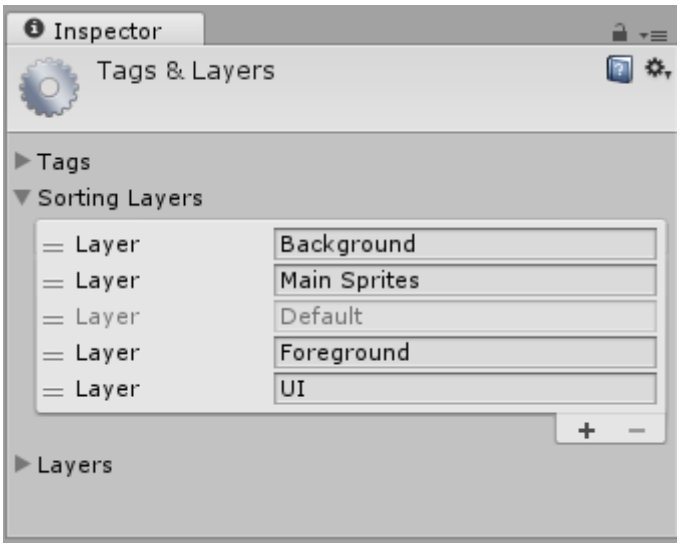
The tags list showing four custom tags

Sorting Layers: Used in conjunction with Sprite graphics in the 2D system, "sorting" refers to the overlay order of different Sprites.



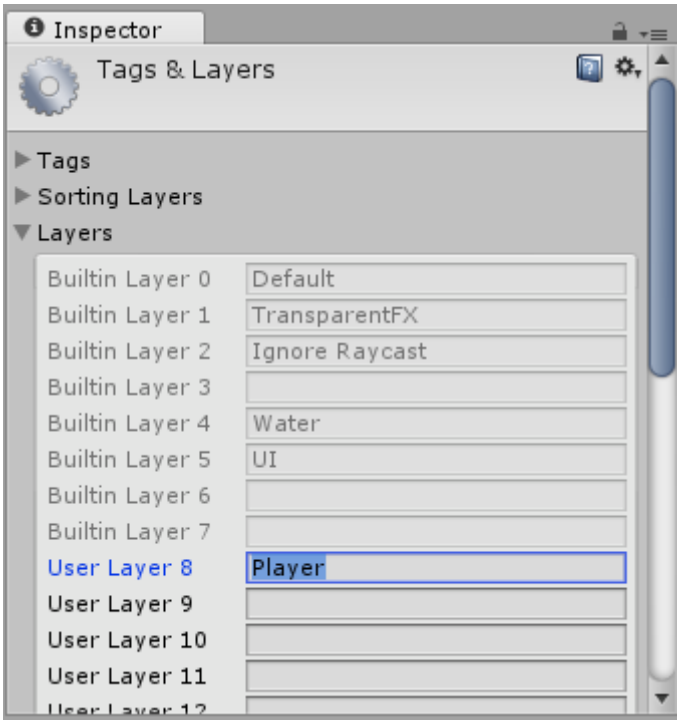
Adding a new Sorting Layer

To add and remove Sorting Layers, use the plus and minus (+/-) buttons at the bottom-right of the list. To change their order, drag the handle at the left-hand side of each Layer item.



The **Sorting Layers** list, showing four custom sorting layers

Layers: Use these throughout the Unity Editor as a way to create groups of objects that share particular characteristics (see documentation on Layers for further details). User Layers primarily to restrict operations such as raycasting or **rendering**, so that they are only applied to the relevant groups of objects. In the Tags and Layers Manager, the first eight **Builtin Layers** are defaults used by Unity, so you cannot edit them. However, you can customise **User Layers** from 8 to 31.



Adding a new Layer

To customise **User Layers** from 8 to 31; type a custom name into the text field for each one you wish to use. Note that you can't add to the number of Layers but, unlike Tags, you can rename Layers.

Audio Manager

Leave feedback

The **Audio Manager** allows you to tweak the maximum volume of all sounds playing in the **scene**. To see it choose **Edit > Project Settings > Audio**.

Properties

Property:	*Function:*
Volume	The volume of all sounds playing.
Rolloff Scale	Sets the global attenuation rolloff factor for Logarithmic rolloff based sources (see Audio Source). The higher the value the faster the volume will attenuate, conversely the lower the value, the slower it attenuate (value of 1 will simulate the "real world").
Doppler Factor	How audible the Doppler effect is. When it is zero it is turned off. 1 means it should be quite audible for fast moving objects.
Default Speaker Mode	Defines which speaker mode should be the default for your project. Default is 2 for stereo speaker setups (see AudioSpeakerMode in the scripting API reference for a list of modes).
Sample Rate	Output sample rate. If set to 0, the sample rate of the system will be used. Also note that this only serves as a reference as only certain platforms allow changing this, such as **iOS** or Android.
DSP Buffer Size	The size of the DSP buffer can be set to optimise for latency or performance
Default	Default buffer size
Best Latency	Trades off performance in favour of latency
Good Latency	Balance between latency and performance
Best Performance	Trades off latency in favour of performance
Virtual Voice Count	Number of virtual voices that the audio system manages. This value should always be larger than the number of voices played by the game. If not, warnings will be shown in the console.
Real Voice Count	Number of real voices that can be played at the same time. Every frame the loudest voices will be picked.
Disable Audio	Deactivates the audio system in standalone builds. Note that this also affects the audio of MovieTextures. In the editor the audio system is still on and will support previewing **audio clips**, but AudioSource. Play calls and playOnAwake will not be handled in order to simulate behavior of the standalone build.

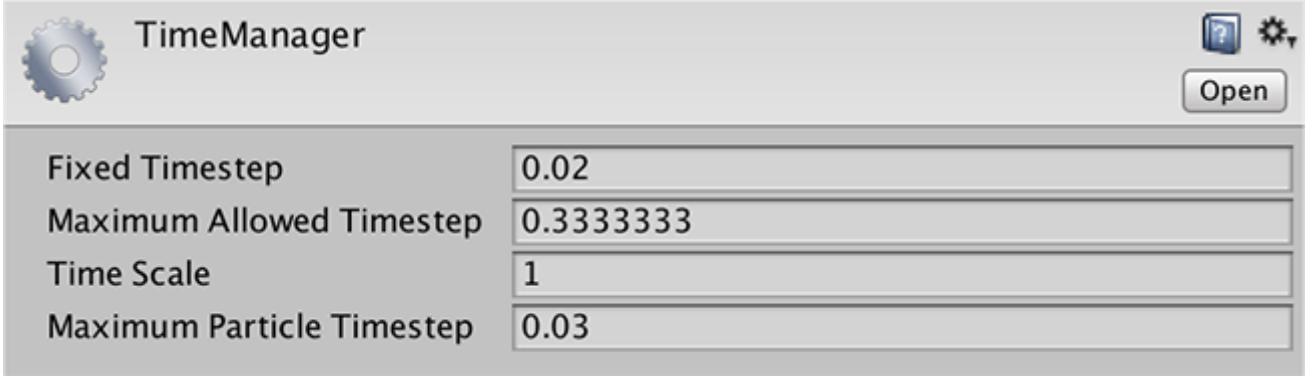
Details

If you want to use Doppler Effect set **Doppler Factor** to 1. Then tweak both **Speed of Sound** and **Doppler Factor** until you are satisfied. Speaker mode can be changed runtime from your application through scripting. See Audio Settings.

Time Manager

Leave feedback

The **Time Manager** (menu: **Edit > Project Settings > Time**) lets you set a number of properties that control timing within your game.



Properties

Property:	*Function:*
Fixed Timestep	A framerate-independent interval that dictates when physics calculations and **FixedUpdate()** events are performed.
Maximum Allowed Timestep	A framerate-independent interval that caps the worst case scenario when frame-rate is low. Physics calculations and **FixedUpdate()** events will not be performed for longer time than specified.
Time Scale	The speed at which time progresses. Change this value to simulate bullet-time effects. A value of 1 means real-time. A value of .5 means half speed; a value of 2 is double speed.
Maximum Particle Timestep	A framerate-independent interval that controls the accuracy of the particle simulation. When the frame time exceeds this value, multiple iterations of the particle update are performed in one frame, so that the duration of each step does not exceed this value. For example, a game running at 30fps (0.03 seconds per frame) could run the particle update at 60fps (in steps of 0.0167 seconds) to achieve a more accurate simulation, at the expense of performance.

Details

The Time Manager lets you set properties globally, but it is often useful to set them from a script during gameplay (for example, setting *Time Scale* to zero is a useful way to pause the game). See the page on Time and Framerate Management for full details of how time can be managed in Unity.

2017–05–18 Page published with limited editorial review

Maximum Particle Timestep added in 2017.1

Player Settings

Leave feedback

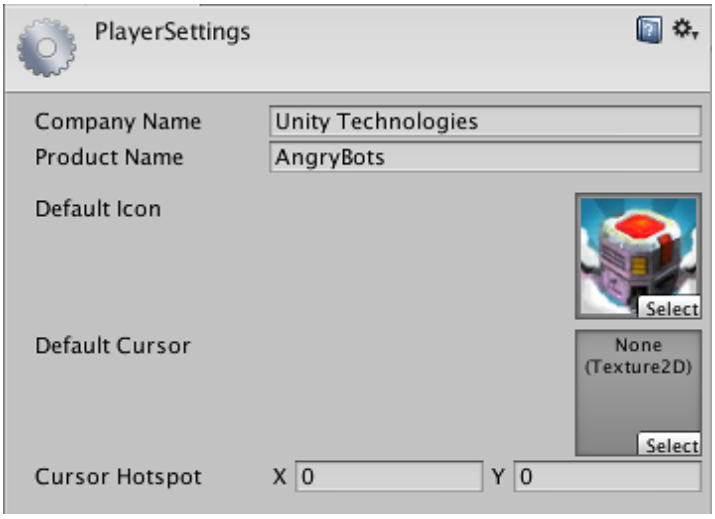
SWITCH TO SCRIPTING

The **Player Settings** (menu: **Edit > Project Settings > Player**) let you set various options for the final game built by Unity. There are a few settings that are the same regardless of the build target but most are platform-specific and divided into the following sections:

Resolution and Presentation: settings for screen resolution and other presentation details such as whether the game should default to fullscreen mode.
Icon: the game icon(s) as shown on the desktop.
Splash Image: the image shown while the game is launching.
Other Settings: any remaining settings specific to the platform.
Publishing Settings: details of how the built application is prepared for delivery from the app store or host webpage.
The general settings are covered below. Settings specific to a platform can be found separately in the platform's own manual section.

See also Unity Splash Screen settings.

General Settings



Property:	*Function:*
Cross-Platform Properties	
Company Name	The name of your company. This is used to locate the preferences file.
Product Name	The name that will appear on the menu bar when your game is running and is used to locate the preferences file also.
Default Icon	The default icon that the application will have on every platform. You can override this for specific platforms.
Default Cursor	The default cursor that the application will have on every supported platform. The supported platforms are PC, MAC and Linux standalone, **WebGL**, **PS4**, and Windows store.

Property:	*Function:*
Cursor Hotspot	Cursor hotspot in **pixels** from the top left of the default cursor.

Splash Screen

Leave feedback

The Unity Editor allows you to configure a Splash Screen for your project. The level to which you can customize the Unity Splash Screen depends on your Unity license; depending on which licence you have, you can disable the Unity Splash Screen entirely, disable the Unity logo, and add your own logos, among other options.

You can also make your own introductory screens or animations to introduce your Project in your first Scene, using the Unity UI System. These can be in addition to or instead of using the Unity Splash Screen, depending on your license.

The Unity Splash Screen is uniform across all platforms. It displays promptly, displaying while the first Scene loads asynchronously in the background. This is different to your own introductory screens or animations which can take time to appear; this is due to Unity having to load the entire engine and first Scene before displaying them.

License limitations

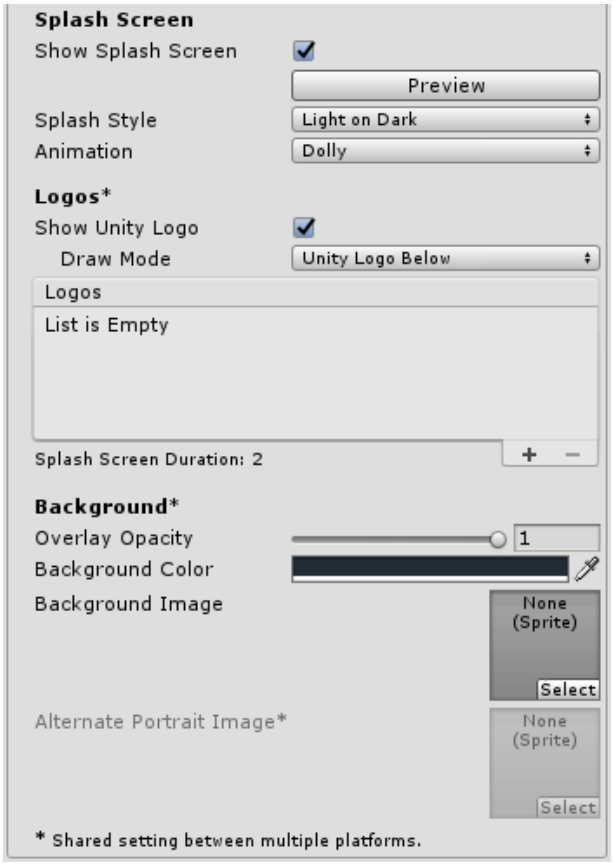
The Unity Pro Edition and Plus Editions licenses have no limitations to customisation of the Unity Splash Screen.

The Unity Personal Edition license has the following limitations:

The Unity Splash Screen cannot be disabled.
The Unity logo cannot be disabled.
The opacity level can be set to a minimum value of 0.5.

Unity Splash Screen settings

To access the Unity Splash Screen settings, go to **Edit** > **Project Settings** > **Player**. In the Inspector window, navigate to **Splash Image** > **Splash Screen**.



Splash Screen settings

Property	Function

Property	Function
Show Splash Screen	Tick the **Show Splash Screen** checkbox to enable the Splash Screen at the start of the game. In the Unity Personal Edition you cannot disable this option; the checkbox is always ticked.
Preview	Use the **Preview** button to see a preview of the Splash Screen in the Game view. The preview reflects the resolution and aspect ratio of the Game view. Use multiple Game views to preview multiple different resolutions and aspect ratios simultaneously. This is particularly useful for simulating the Splash Screen's appearance on multiple different devices. See *Image A*, below, for an example.
Splash Style	**Splash Style** controls the style of the Unity branding. There are two options available: Light on Dark, or Dark on White. See these in *Image B*, below.
Animation	The Splash Screen has 3 possible animation modes, which define how it appears and disappears from the screen.
Static	The Splash Screen has no animation.
Dolly	The logo and background zoom to create a visual dolly effect.
Custom	Configure the background and logo zoom amounts to allow for a modified dolly effect.
Show Unity logo	Tick the **Show Unity Logo** checkbox to enable Unity co-branding. In the Unity Personal Edition you cannot disable this option; the checkbox is always ticked.
Draw Mode	**Draw Mode** controls how Unity co-branding is shown (if Unity co-branding is enabled).
Unity Logo Below	Draws the co-branding Unity logo beneath all logos that are shown.
All Sequential	Inserts the co-branding Unity logo as a logo into the **Logos** list.
Logos	This is the customisable list of logos to be drawn during the duration of the Splash Screen. See the Logo list in *Image C*, below. Add and remove logos using the plus (+) and minus (-) buttons, and reorder logos in the list by dragging and dropping. Each logo must be a Sprite Asset. To change the aspect ratio of the logo, change the dimensions of the Sprite using the Sprite Editor, with **Sprite Mode** set to **Multiple**. The **Logo Duration** of the Sprite Asset is the length of time the logo appears for. This can be set to between a minimum of 2 seconds and a maximum of 10 seconds. If an entry in the Logos list has no logo Sprite Asset assigned, no logo is shown for the duration of that entry. You can use this to create delays between logos. The entire duration of the Splash Screen is the total of all logos plus 0.5 seconds for fading out. This might be longer if the first Scene is not ready to play, in which case the Splash Screen shows only the background image or color and then fades out when the first Scene is ready to play.
Overlay Opacity	Adjust the strength of the **Overlay Opacity** to make the logos stand out; this affects the background color and/or image color, based on the **Splash Style**. Set **Overlay Opacity** to a lower number to reduce this effect, or set it to 0 to disable the effect completely. For example, if the Splash Screen style is **Light on Dark,** with a white background, the background becomes gray when **Overlay Opacity** is set to 1, and white when **Overlay Opacity** is set to 0. In the Unity Personal Edition, this option has a minimum value of 0.5.
Background Color	Use this to set the background color when no background image is set. Note that the actual background color may be affected by the **Overlay Opacity** (see section above), and might not match the assigned color.

Property	Function
	Use this to set a background Sprite image instead of using a color background. Unity adjusts the background image so that it fills the screen; the image is uniformly scaled until it fits both the width and height of the screen. This means that parts of the image might extend beyond the screen edges in some aspect ratios. To adjust the background image's response to aspect ratio, change the Sprite's **Position** values in the Sprite Editor.
Background Image	Use **Alternate Portrait Image** to set an image with portrait aspect ratios (for example, a mobile device in portrait mode). If there is no **Alternate Portrait Image** Sprite assigned, the Unity Editor uses the Sprite assigned as the **Background Image** for both portrait and landscape mode. Adjust the **Position** and dimensions of the Sprite in the Sprite Editor to control the aspect ratio and position of the background image on the Splash Screen. In *Image D*, below, the same image is being used for both landscape and portrait; however, the portrait position has been adjusted.

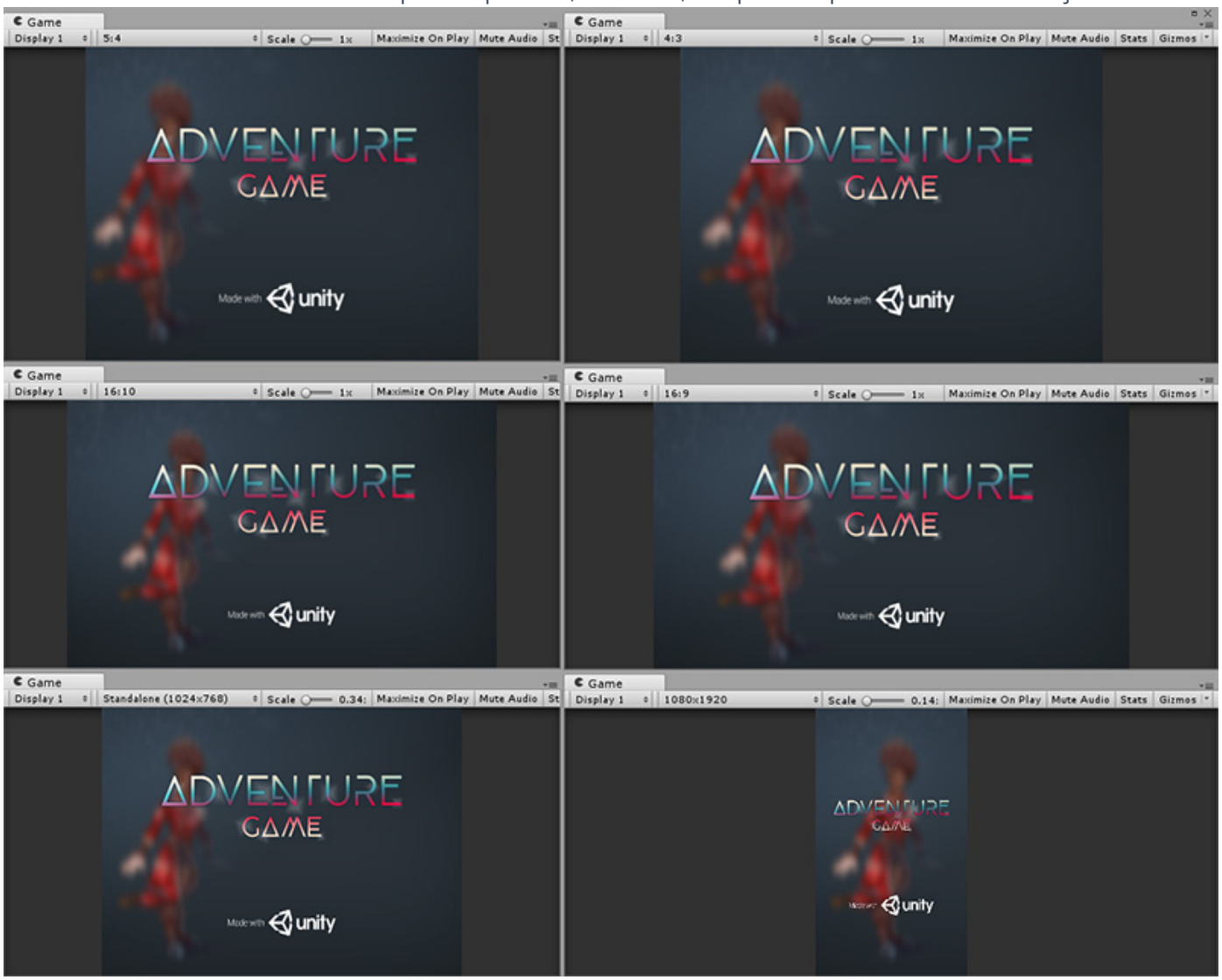


Image A: **Preview** - Multiple previews

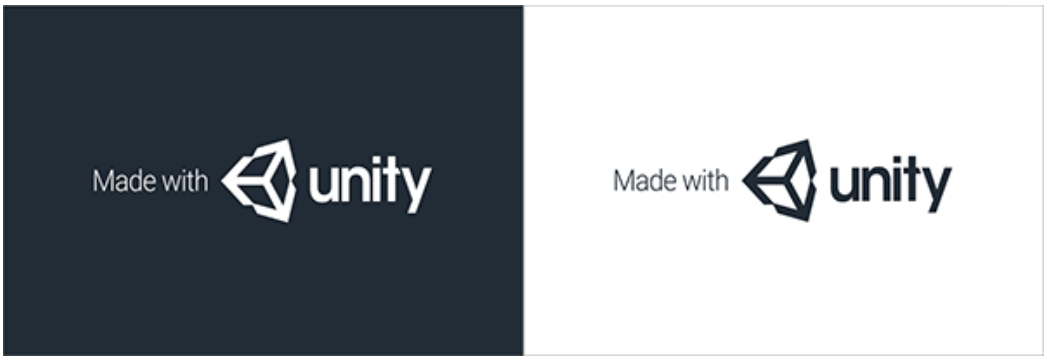


Image B: **Splash Style** - On the left, Light on Dark style. On the right, Dark on Light style.

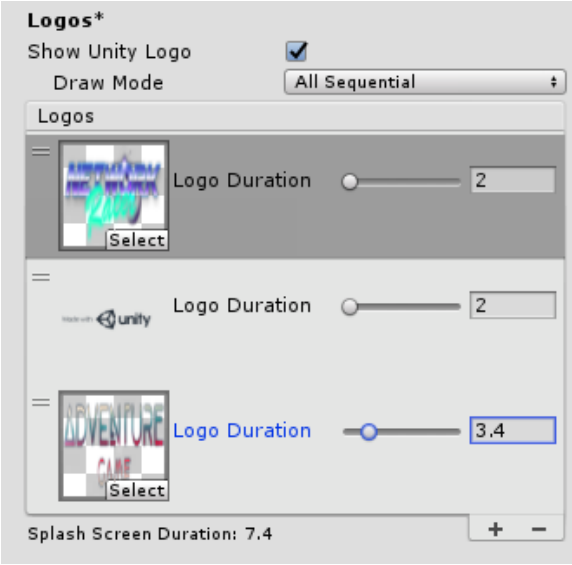


Image C: **Logos** - The Logo list and **Logo Duration**

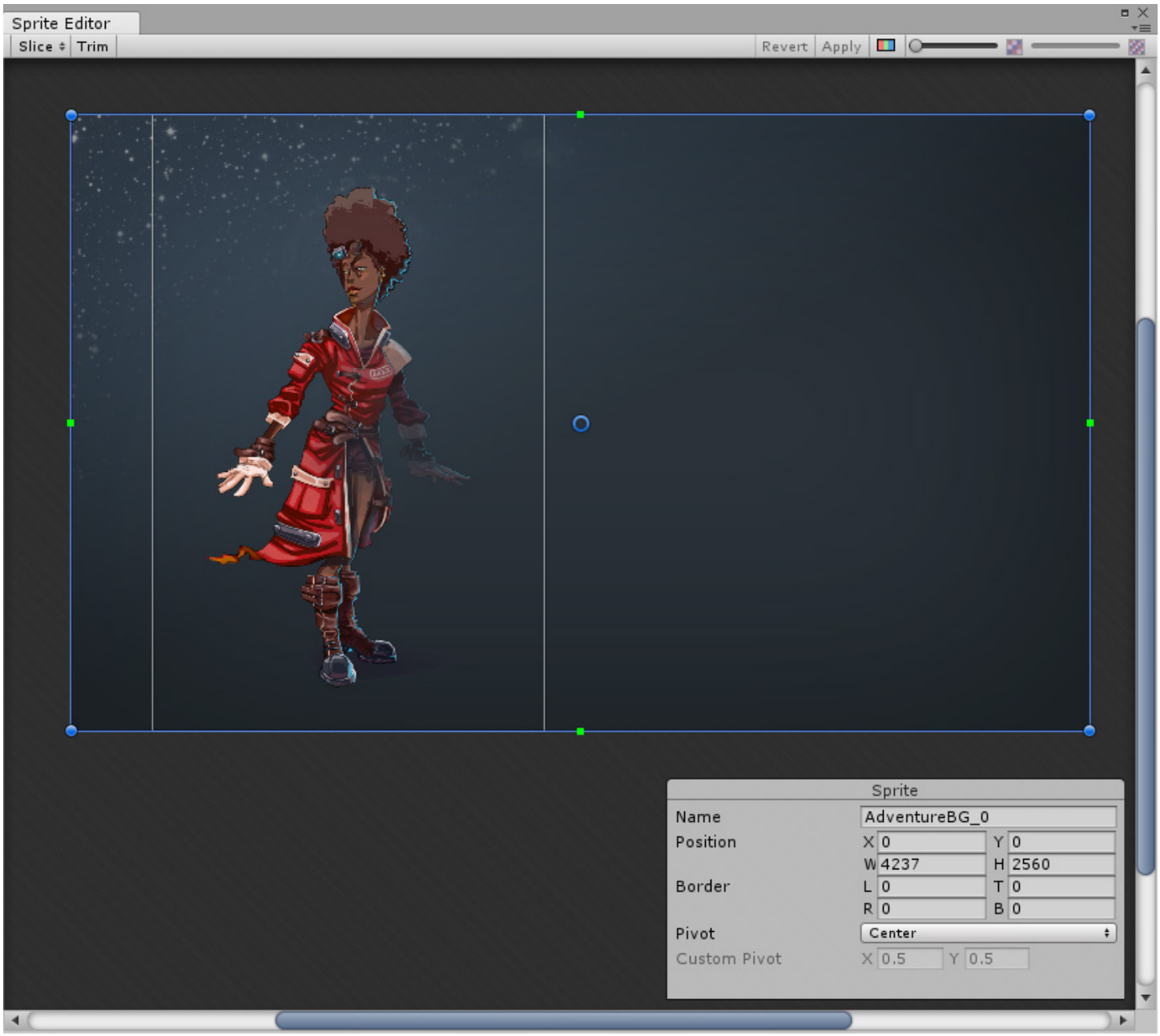


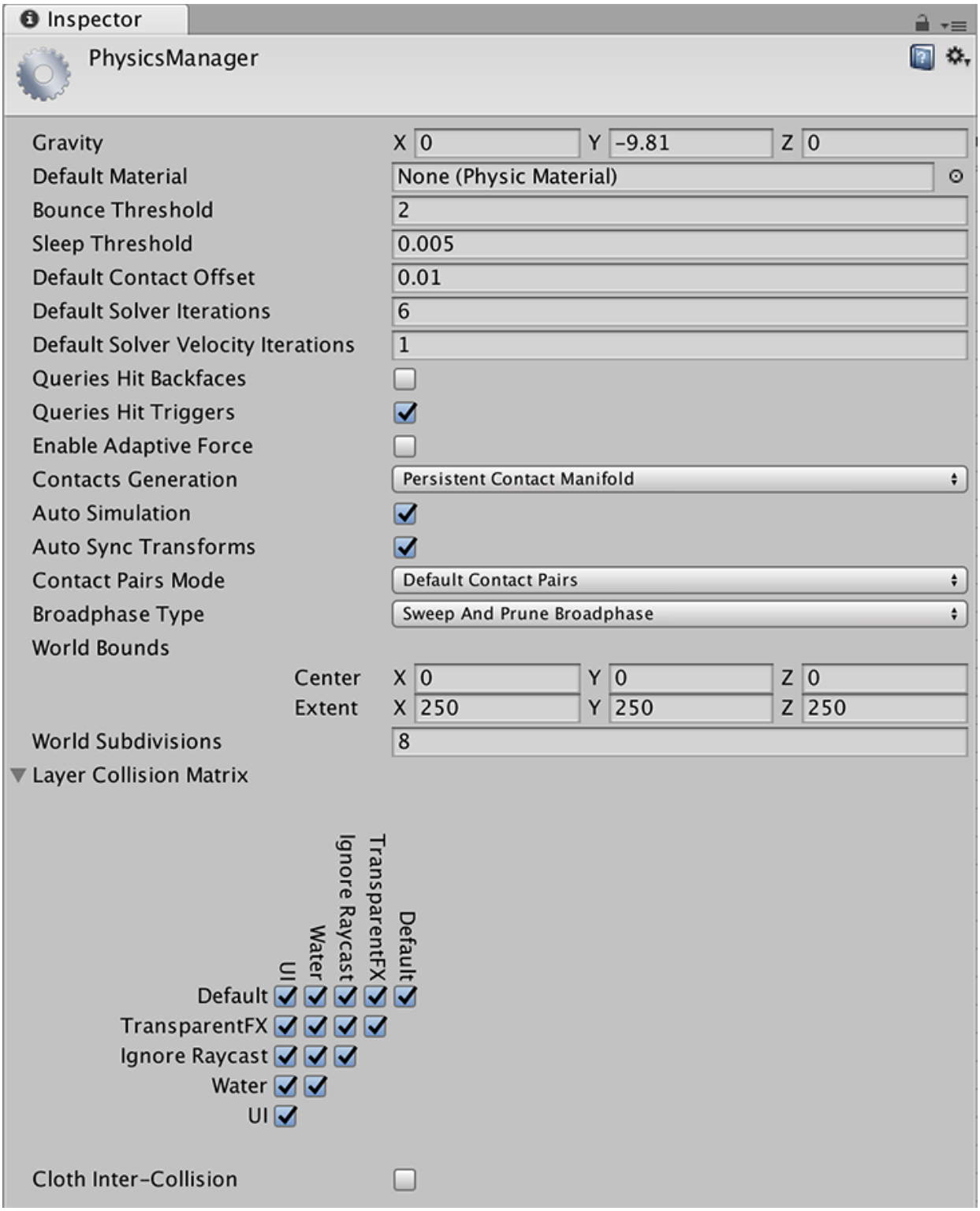
Image D: **Background Image** - The same is being used here for both landscape and portrait

Physics Manager

Leave feedback

Use the Physics Manager to apply global settings for 3D physics. To access it, go to **Edit** > **Project Settings** > **Physics**. To manage global settings for 2D physics, use the Physics 2D Manager.

The settings of the PhysicsManager define limits on the accuracy of the physical simulation. Generally speaking, a more accurate simulation requires more processing overhead, so these settings offer a way to trade off accuracy against performance. See the Physics section of the manual for further information.



The PhysicsManager, shown in the Inspector window.

Property: **Function:**

Property:	Function:
Gravity	Use the x, y and z axes to set the amount of gravity applied to all Rigidbody components. For realistic gravity settings, apply a negative number to the y axis. Gravity is defined in world units per seconds squared. **Note**: If you increase the gravity, you might need to also increase the **Default Solver Iterations** value to maintain stable contacts.
Default Material	Use this field to define the default Physics Material that is used if none has been assigned to an individual **Collider**.
Bounce Threshold	Use this to set a velocity value. If two colliding objects have a relative velocity below this value, they do not bounce off each other. This value also reduces jitter, so it is not recommended to set it to a very low value.
Sleep Threshold	Use this to set a global energy threshold, below which a non-kinematic **Rigidbody** (that is, one that is not controlled by the physics system) may go to sleep. When a Rigidbody is sleeping, it is not updated every frame, making it less resource-intensive. If a Rigidbody's kinetic energy divided by its mass is below this threshold, it is a candidate for sleeping.
Default Contact Offset	Use this to set the distance the **collision detection** system uses to generate **collision** contacts. The value must be positive, and if set too close to zero, it can cause jitter. This is set to 0.01 by default. Colliders only generate **collision** contacts if their distance is less than the sum of their contact offset values.
Default Solver Iterations	Solvers are small **physics engine** tasks which determine a number of physics interactions, such as the movements of **joints** or managing contact between overlapping Rigidbody **components**. Use **Default Solver Iterations** to define how many solver processes Unity runs on every physics frame. This affects the quality of the solver output and it's advisable to change the property in case non-default `Time.fixedDeltaTime` is used, or the configuration is extra demanding. Typically, it's used to reduce the jitter resulting from joints or contacts.
Default Solver Velocity Iterations	Use this to set how many velocity processes a solver performs in each physics frame. The more processes the solver performs, the higher the accuracy of the resulting exit velocity after a Rigidbody bounce. If you experience problems with jointed Rigidbody components or Ragdolls moving too much after collisions, try increasing this value.
Queries Hit Backfaces	Tick the checkbox if you want physics queries (such as `Physics.Raycast`) to detect hits with the backface triangles of MeshColliders. This setting is unticked by default.
Queries Hit Triggers	Tick the checkbox if you want physics hit tests (such as Raycasts, SphereCasts and SphereTests) to return a hit when they intersect with a Collider marked as a Trigger. Individual raycasts can override this behavior. This setting is ticked by default.
Enable Adaptive Force	The adaptive force affects the way forces are transmitted through a pile or stack of objects, to give more realistic behaviour. Tick this box to enable the adaptive force. This setting is unticked by default.
Contacts Generation	Choose a contact generation method: **Persistent Contacts Manifold (PCM)**: Regenerates fewer contacts every physics frame and shares more contact data across frames. The PCM contacts generation path is also more accurate, and usually produces better collision feedback in most of the cases. See Nvidia documentation on Persistent Contact Manifold for more information. **Legacy Contacts Generation**: Generates contacts using the separating axis theorem (see dyn4j.org's guide to SAT). PCM is more efficient, but for projects made in versions of Unity earlier than Unity 5.5, you might find it easier to continue using SAT, to avoid needing to retweak physics slightly. PCM can result in a slightly different bounce, and fewer useless contacts end up in the contacts buffers (that is, the arrays you get in the Collision instance passed to `OnCollisionEnter`, `OnCollisionStay`, and `OnCollisionExit`).
Auto Simulation	Run the physics simulation automatically or allow explicit control over it

Property:	Function:
Auto Sync Transforms	Automatically sync transform changes with the physics system whenever a **Transform component** changes.
Contact Pairs Mode	Select the type of contact pair generation to use: **Default Contact Pairs**: Receive collision and trigger events from all contact pairs except kinematic-kinematic and kinematic-static pairs. **Enable Kinematic Kinematic Pairs**: Receive collision and trigger events from kinematic-kinematic contact pairs. **Enable Kinematic Static Pairs**: Receive collision and trigger events from kinematic-static contact pairs. **Enable All Contact Pairs**:Receive collision and trigger events from all contact pairs.
Broadphase Type	Choose which broad-phase algorithm to use in the physics simulation: **Sweep and Prune Broadphase** or **Multibox Pruning Broadphase**. Multi-box pruning uses a grid, and each grid cell performs sweep-and-prune. This usually helps improve performance if, for example, there are many **GameObjects** in a flat world. For more information, see NVIDIA's documentation on PhysX SDK and Rigid Body Collision.
World Bounds	Define the 2D grid that encloses the world to prevent far away objects from affecting each other when using the **Multibox Pruning Broadphase** algorithm. This option is only available when you set **Broadphase Type** to **Multibox Pruning Broadphase**.
World Subdivisions	The number of cells along the x and z axis in the 2D gri__ algorithm. This option is only available when you set **Broadphase Type** to **Multibox Pruning Broadphase**.
Layer Collision Matrix	Use this to define how the layer-based collision detection system behaves.
Cloth Inter-Collision	Control cloth particle **Distance** and **Stiffness** when setting up particles for Cloth intercollision.

2018–08–10 Page amended with limited editorial review

Settings added in 5.5: Default Solver Iterations, Default Solver Velocity Iterations, Queries Hit Backfaces, Enable PCM

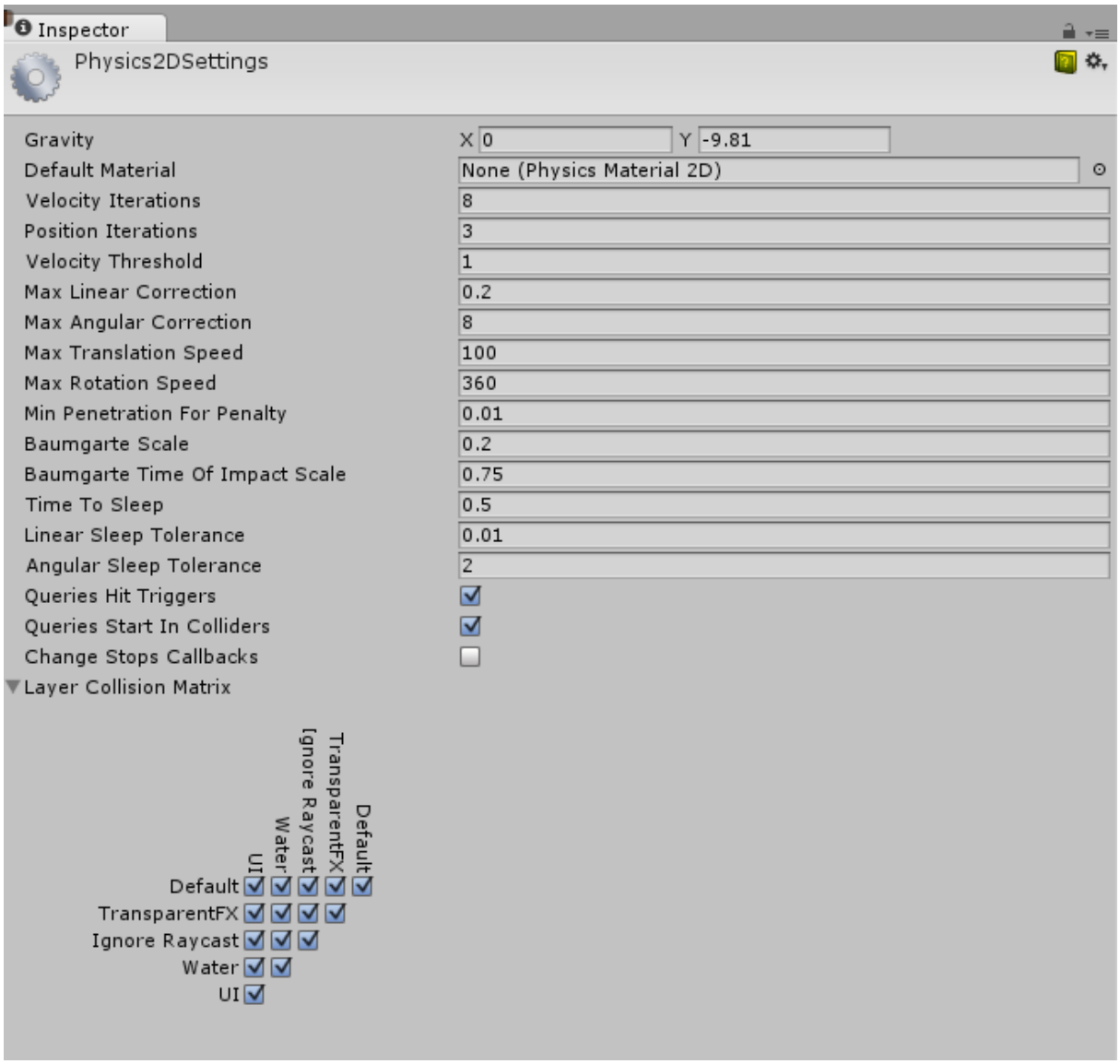
Physics Manager settings documentation updated in 2017.4

Physics 2D Settings

Leave feedback

The **Physics 2D Settings** allow you to provide global settings for 2D physics (menu: **Edit** > **Project Settings** > **Physics 2D**).

(There is also a corresponding Physics Manager for 3D projects.)



Properties

Property:	*Function:*
Gravity	The amount of gravity applied to all Rigidbody 2D **GameObjects**. Generally, gravity is only set for the negative direction of the y-axis.
Default Material	The default Physics Material 2D that is used if none has been assigned to an individual **Collider** 2D.
Velocity Iterations	The number of iterations made by the **physics engine** to resolve velocity effects. Higher numbers result in more accurate physics but at the cost of CPU time.
Position Iterations	The number of iterations made by the physics engine to resolve position changes. Higher numbers result in more accurate physics but at the cost of CPU time.
Velocity Threshold	**Collisions** with a relative velocity lower than this value is treated as inelastic **collisions** (that is, the colliding GameObjects do not bounce off each other).

Property:	*Function:*
Max Linear Correction	The maximum linear position correction used when solving constraints (from a range between 0.0001 to 1000000). This helps to prevent overshoot.
Max Angular Correction	The maximum angular correction used when solving constraints (froma range between 0.0001 to 1000000). This helps to prevent overshoot.
Max Translation Speed	The maximum linear speed of a **Rigidbody** 2D GameObject during any physics update.
Max Rotation Speed	The maximum rotation speed of a Rigidbody 2D GameObject during any physics update.
Min Penetration For Penalty	The minimum contact penetration radius allowed before any separation impulse force is applied.
Baumgarte Scale	Scale factor that determines how fast collision overlaps are resolved.
Baumgarte Time of Impact Scale	Scale factor that determines how fast time-of-impact overlaps are resolved.
Time to Sleep	The time (in seconds) that must pass after a Rigidbody 2D stops moving before it goes to sleep.
Linear Sleep Tolerance	The linear speed below which a Rigidbody 2D goes to sleep after the **Time to Sleep** elapses.
Angular Sleep Tolerance	The rotational speed below which a Rigidbody 2D goes to sleep after **Time to Sleep** elapses.
Queries Hit Triggers	Check this box to enable Collider 2Ds marked as **Triggers** to return a hit when any physics queries (such as Linecasts or Raycasts) intersect with them. Leave it unchecked for these queries to not return a hit.
Queries Start In Colliders	Check this box to enable physics queries that start inside a Collider 2D to detect the collider they start in.
Change Stops Callbacks	Check this box to stop reporting collision callbacks immediately if any of the GameObjects involved in the collision are deleted or moved.
Layer Collision Matrix	Defines how the Layer-based collision detection system behaves.

Notes

The Physics 2D Settings define limits on the accuracy of the physical simulation. Generally speaking, a more accurate simulation requires more processing overhead, so these settings offer a way to trade off accuracy against performance. See the Physics section of the manual for further information.

Quality Settings

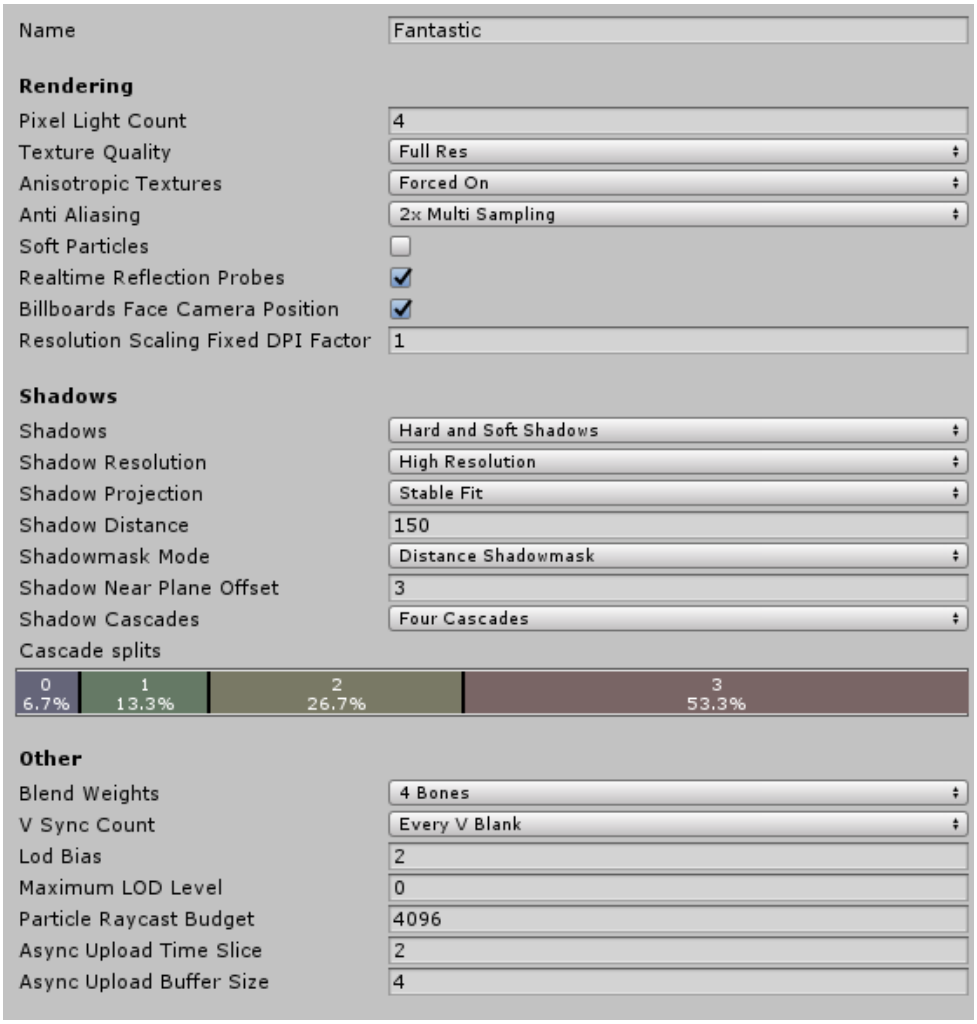
Leave feedback

SWITCH TO SCRIPTING

Unity allows you to set the level of graphical quality it will attempt to render. Generally speaking, quality comes at the expense of framerate and so it may be best not to aim for the highest quality on mobile devices or older hardware since it will have a detrimental effect on gameplay. The **Quality Settings** inspector (menu: **Edit > Project Settings > Quality**) is used to select the quality level in the editor for the chosen device. It is split into two main areas - at the top, there is the following matrix:

Unity lets you assign a name to a given combination of quality options for easy reference. The rows of the matrix let you choose which of the different platforms each quality level will apply to. The Default row at the bottom of the matrix is not a quality level in itself but rather sets the default quality level used for each platform (a green checkbox in a column denotes the level currently chosen for that platform). Unity comes with six quality levels pre-enabled but you can add your own levels using the button below the matrix. You can use the trashcan icon (the rightmost column) to delete an unwanted quality level.

You can click on the name of a quality level to select it for editing, which is done in the panel below the settings matrix:



The quality options you can choose for a quality level are as follows:

Property:	*Function:*
Name	The name that will be used to refer to this quality level

Rendering

Property:	*Function:*
Pixel Light Count	The maximum number of **pixel** lights when **Forward Rendering** is used.
Texture Quality	This lets you choose whether to display textures at maximum resolution or at a fraction of this (lower resolution has less processing overhead). The options are **Full Res**, **Half Res**, **Quarter Res** and **Eighth Res**.
Anisotropic Textures	This enables if and how anisotropic textures will be used. The options are **Disabled**, **Per Texture** and **Forced On** (ie, always enabled).
AntiAliasing	This sets the level of antialiasing that will be used. The options are **2x**, **4x** and **8x** multi-sampling.
Soft Particles	Should soft blending be used for particles?
Realtime Reflection Probes	Should reflection probes be updated during gameplay?
Resolution Scaling Fixed DPI Factor	Downscales the device's screen resolution below its native resolution. For more details, see the platform-specific **Player Settings** pages, such as Android Player Settings and iOS Player Settings.

Shadows

Property:	*Function:*

Property:	*Function:*
Shadows	This determines which type of shadows should be used. The available options are **Hard and Soft Shadows**, **Hard Shadows Only** and **Disable Shadows**.
Shadow resolution	Shadows can be rendered at several different resolutions: **Low**, **Medium**, **High** and **Very High**. The higher the resolution, the greater the processing overhead.
Shadow Projection	There are two different methods for projecting shadows from a directional light. **Close Fit** renders higher resolution shadows but they can sometimes wobble slightly if the **camera** moves. **Stable Fit** renders lower resolution shadows but they don't wobble with **camera** movements.
Shadow Cascades	The number of shadow cascades can be set to zero, two or four. A higher number of cascades gives better quality but at the expense of processing overhead (see Directional Light Shadows for further details).
Shadow Distance	The maximum distance from camera at which shadows will be visible. Shadows that fall beyond this distance will not be rendered.
Shadowmask Mode	Sets the shadowmask behaviour when using the Shadowmask Mixed lighting mode. Use the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**) to set this up in your Scene.
Distance Shadowmask	Unity uses real-time shadows up to the **Shadow Distance**, and baked shadows beyond it.
Shadowmask	hadowmask: Static **GameObjects** that cast shadows always cast baked shadows.
Shadow Near Plane Offset	Offset shadow near plane to account for large triangles being distorted by shadow pancaking.

Other

Property:	*Function:*
Blend Weights	The number of bones that can affect a given vertex during an animation. The available options are one, two or four bones.
VSync Count	Rendering can be synchronised with the refresh rate of the display device to avoid "tearing" artifacts (see below). You can choose to synchronise with every vertical blank (VBlank), every second vertical blank or not to synchronise at all.
LOD Bias	**LOD** levels are chosen based on the onscreen size of an object. When the size is between two **LOD** levels, the choice can be biased toward the less detailed or more detailed of the two models available. This is set as a fraction from 0 to +infinity. When it is set between 0 and 1 it favors less detail. A setting of more than 1 favors greater detail. For example, setting **LOD** Bias to 2 and having it change at 50% distance, **LOD** actually only changes on 25%.
Maximum LOD Level	The highest LOD that will be used by the game. See note below for more Information.
Particle Raycast Budget	The maximum number of raycasts to use for approximate **particle system collisions** (those with **Medium** or **Low** quality). See Particle System Collision Module.
Async Upload Time Slice	The amount of CPU time in milliseconds per frame to spend uploading buffered textures to the GPU. See Async Texture Upload.
Async Upload Buffer Size	The size in MB for the Async Upload buffer. See Async Texture Upload.

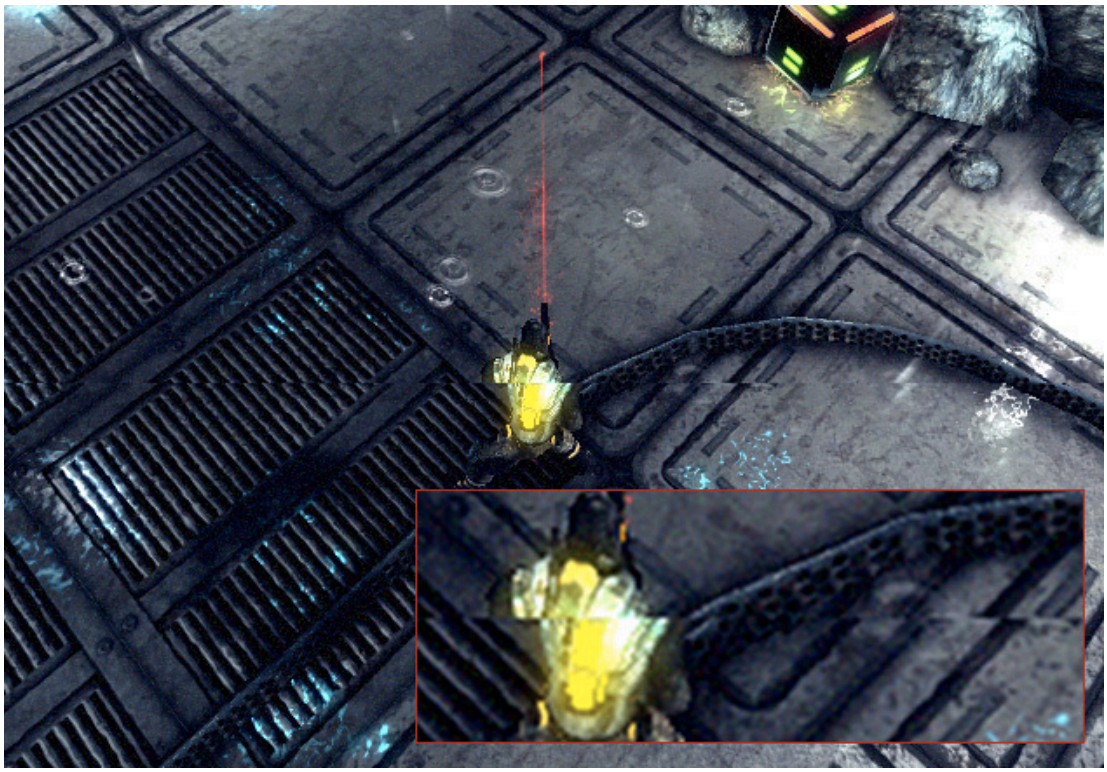
MaximumLOD level

Models which have a LOD below the MaximumLOD level will not be used and omitted from the build (which will save storage and memory space). Unity will use the smallest LOD value from all the MaximumLOD values linked with the quality settings for the target platform. If an LOD level is included then models from that LODGroup will be included in the build and always loaded at

runtime for that LODGroup, regardless of the quality setting being used. As an example, if LOD level 0 is used in any quality setting then all the LOD levels will be included in the build and all the referenced models loaded at runtime.

Tearing

The picture on the display device is not continuously updated but rather the updates happen at regular intervals much like frame updates in Unity. However, Unity's updates are not necessarily synchronised with those of the display, so it is possible for Unity to issue a new frame while the display is still rendering the previous one. This will result in a visual artifact called "tearing" at the position onscreen where the frame change occurs.

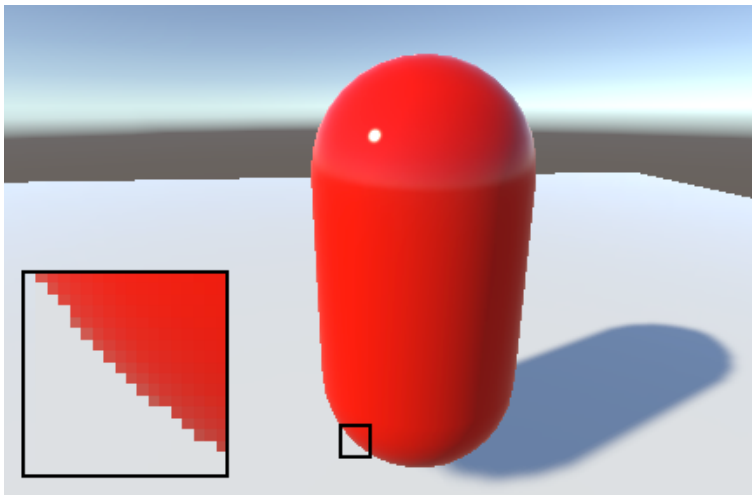


Simulated example of tearing. The shift in the picture is clearly visible in the magnified portion.

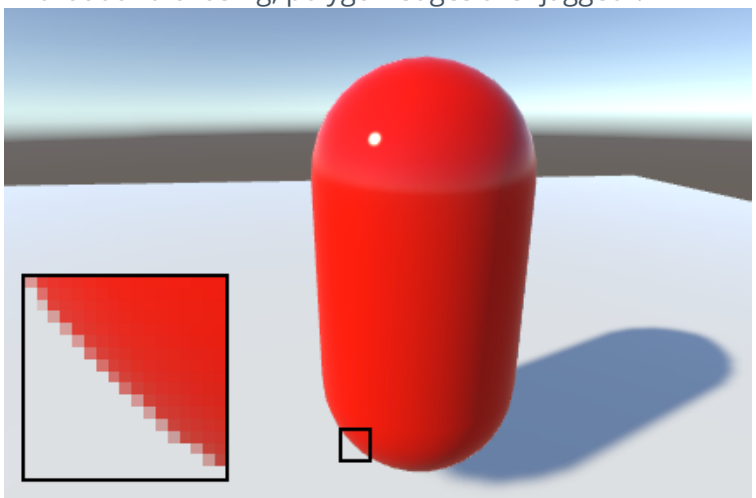
It is possible to set Unity to switch frames only during the period where the display device is not updating, the so-called "vertical blank". The VSync option on the Quality Settings synchronises frame switches with the device's vertical blank or optionally with every other vertical blank. The latter may be useful if the game requires more than one device update to complete the rendering of a frame.

Anti-aliasing

Anti aliasing improves the appearance of polygon edges, so they are not "jagged", but smoothed out on the screen. However, it incurs a performance cost for the graphics card and uses more video memory (there's no cost on the CPU though). The level of anti-aliasing determines how smooth polygon edges are (and how much video memory does it consume).



Without anti-aliasing, polygon edges are “jagged”.

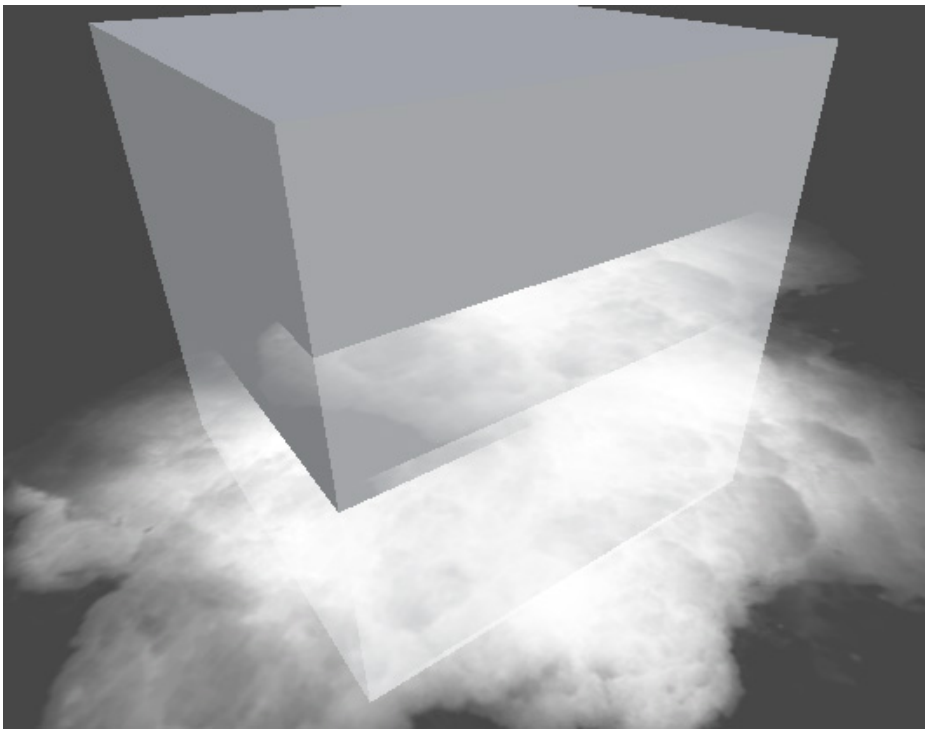


With 4x anti-aliasing, polygon edges are smoothed out.

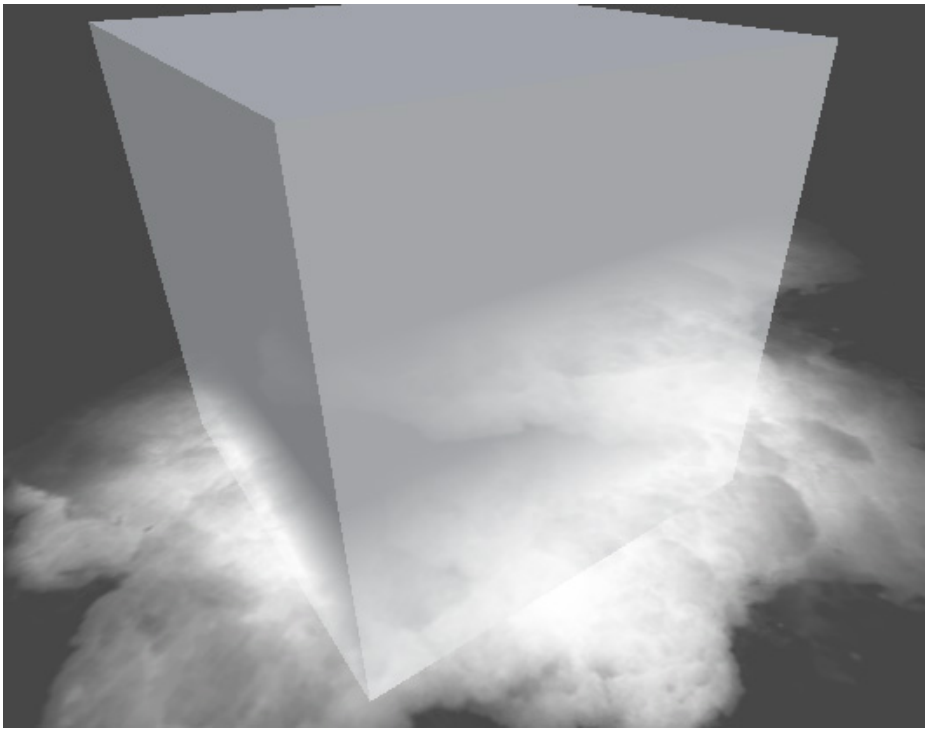
However, built-in hardware anti-aliasing does not work with Deferred Shading or HDR rendering; for these cases you’ll need to use Antialiasing Image Effect.

Soft Particles

Soft Particles fade out near intersections with other **scene** geometry. This looks much nicer, however it’s more expensive to compute (more complex pixel shaders), and only works on platforms that support depth textures. Furthermore, you have to use Deferred Shading or Legacy Deferred Lighting **rendering path**, or make the camera render depth textures from **scripts**.



Without Soft Particles - visible intersections with the scene.



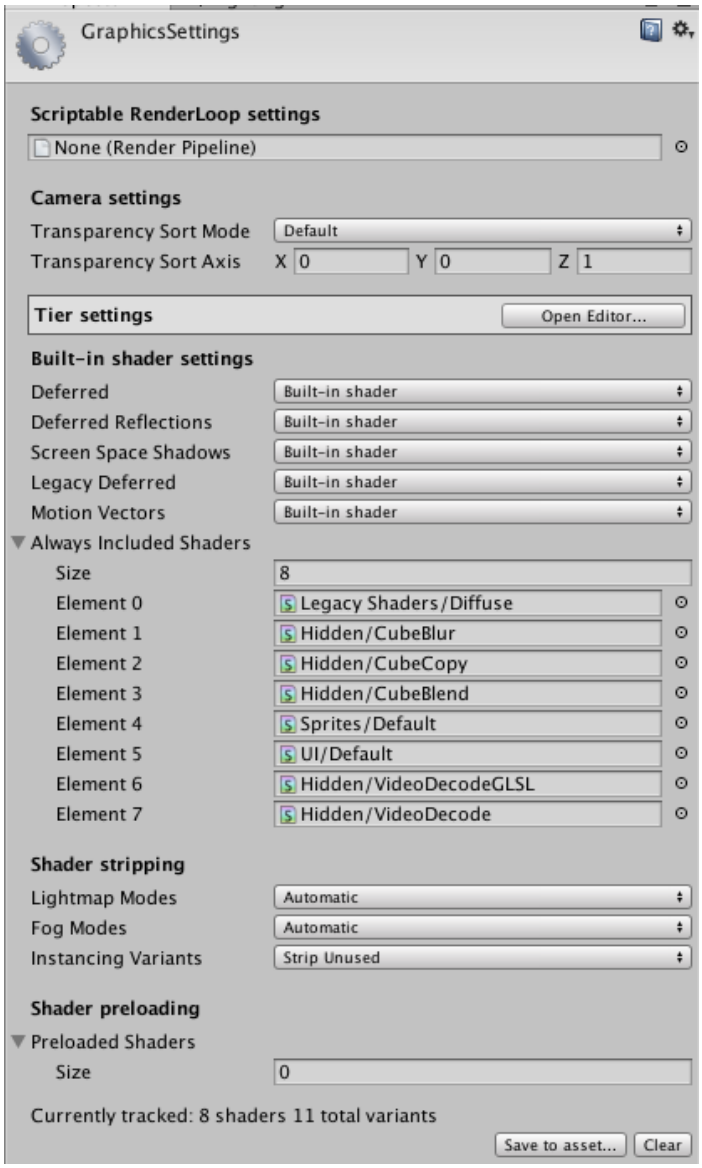
With Soft Particles - intersections fade out smoothly.

2017–09–18 Page amended with limited editorial review

Shadowmask Mode added in 2017.1

Graphics Settings

Leave feedback



Scriptable RenderLoop settings

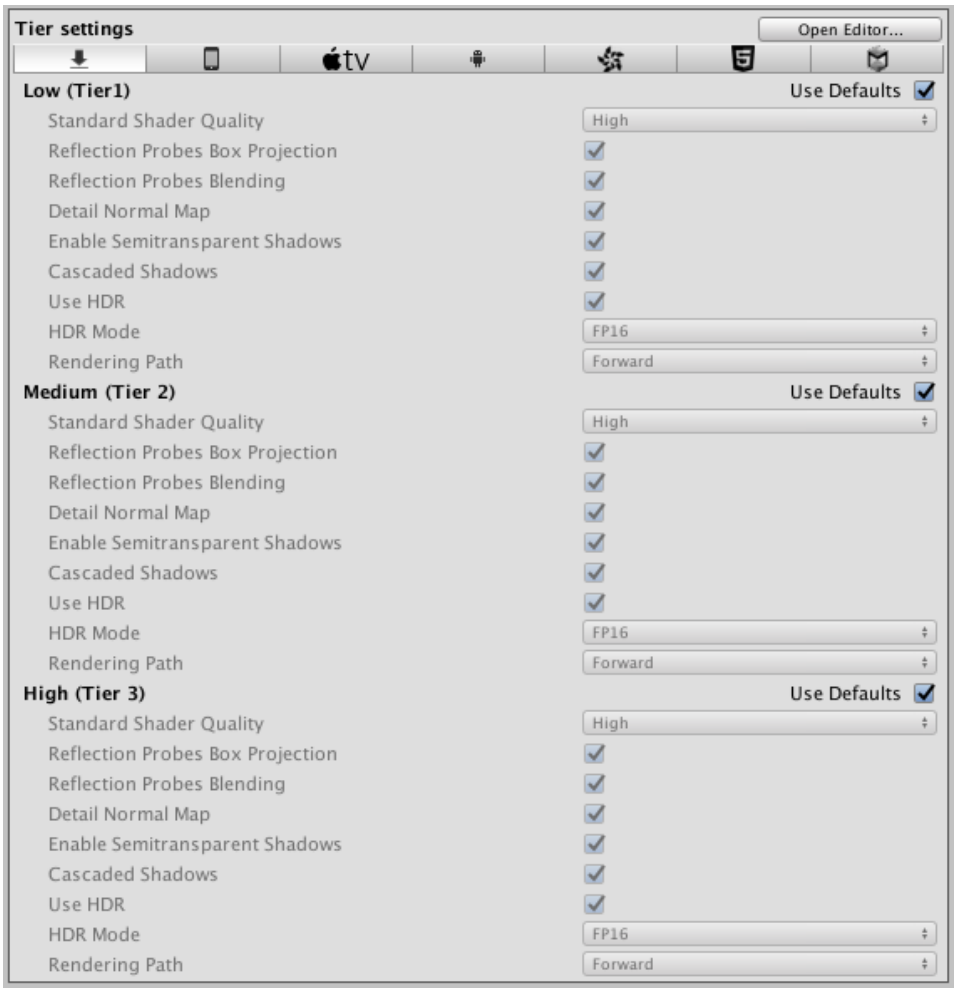
This is an experimental setting which allows you to define a series of commands to control exactly how the **Scene** should be rendered (instead of using the default rendering pipeline used by Unity). For more information on this experimental feature, see the Scriptable Render Pipeline documentation on GitHub.

Camera settings

These properties control various **rendering** settings.

Setting:	**Description:**
Transparancy Sort Mode	Renderers in Unity are sorted by several criteria such as their layer number, or their distance from **camera**. Transparency Sort Mode adds the ability to order renderable objects by their distance along a specific axis. This is generally only useful in 2D development; for example, sorting **Sprites** by height or along the Y axis.
Default	Sorts objects based on the Camera mode.
Perspective	Sorts objects based on perspective view.
Orthographic	Sorts objects based on orthographic view.
Transparancy Sort Axis	Use this to set a custom Transparency Sort Mode.

Tier settings



Tier Settings as displayed in the PlayerSettings Inspector window

These settings allow you to make platform-specific adjustments to rendering and shader compilation, by tweaking builtin defines. For example, you can use this to enable Cascaded Shadows on high-tier **iOS** devices, but to disable them on low-tier devices to improve performance. Tiers are defined by Rendering.GraphicsTier.

Property::	Function:
Standard Shader Quality	Allows you to select **Standard Shader** Quality.
Reflection Probes Box Projection	Allows you to specify whether **Reflection Probes** Box Projection should be used.
Reflection Probes Blending	Allows you to specify whether Reflection Probes Blending should be enabled.
Detail Normal Map	Allows you to specify whether Detail Normal Map should be sampled if assigned.
Enable Semitransparent Shadows	Allows you to specify whether Semitransparent Shadows should be enabled.
Cascaded Shadows	Allows you to specify whether cascaded shadow maps should be used.
Use HDR	Setting this field to true enables **HDR** rendering for this tier. Setting it to false disables **HDR** rendering for this tier. See Also: High Dynamic Range rendering
HDR Mode	The CameraHDRMode to use for this tier.
Rendering Path	The rendering path that should be used.

Built-in shader settings

Use these settings to specify which shader is used to do lighting pass calculations in each rendering path listed.

Shader:	Calculation:
Deferred	Used when using Deferred lighting, see [Camera: Rendering Path](class-Camera)

Shader:	Calculation:
Deferred Reflection	Used when using Deferred reflection (ie reflection probes) along deferred lighting, see [Camera: Rendering Path)(class-Camera_)
Screen Space shadows	Used when using screen space shadow.
Legacy deferred	Used when using Legacy deferred lighting, see [Camera: Rendering Path)(class-Camera).
Motion vectors	Used when using Legacy deferred lighting, see MeshRenderer::Motion Vectors.
Lens Flare	Used when using Lens Flares, see Flare.
Light Halo	Used when using Light Halos, see Halo.

Setting:	Description:
Built-in shader (Default value)	Use Unity's built-in shaders to do the calculation.
Custom shader	Use your own compatible shader to do the calculation. This enables you to do deep customization of deferred rendering.
No Support	Disable this calculation. Use this setting if you are not using **deferred shading** or lighting. This will save some space in the built game data files.

Always-included Shaders

Specify a list of Shaders that will always be stored along with the project, even if nothing in your scenes actually uses them. It is important to add shaders used by streamed AssetBundles to this list to ensure they can be accessed.

Shader stripping - Lightmap modes and Fog modes

Lower your build data size and improve loading times by stripping out certain shaders involved with lighting and fog.

Setting:	*Description:*
Automatic (Default value)	Unity looks at your scenes and lightmapping settings to figure out which fog and lightmapping modes are not in use, and skips corresponding shader variants.
Manual	Specify which modes to use yourself. Select this if you are building asset bundles or changing fog modes from a script at runtime, to ensure that the modes you want to use are included.

Shader stripping - Instancing variants

Setting:	*Description:*
Strip Unused (Default value)	When a project is built, Unity only includes instancing shader variants if at least one material referencing the shader has the "Enable instancing" checkbox ticked. Unity strips any shaders that are not referenced by materials with the "Enable instancing" checkbox ticked.
Strip All	Strip all instancing shader variants, even if they are being used.
Keep All	Keep all instancing shader variants, even if they are not being used.

See GPU instancing for more information about Instancing variants.

Shader preloading

Specify a list of shader variant collection **assets** to preload while loading the game. Shader variants specified in this list are loaded during entire lifetime of the application. Use it for preloading very frequently used shaders. See Optimizing Shader Load Time page for details.

See also

Optimizing Shader Load Time
Optimizing Graphics Performance
Shaders reference

17–05–08 Page amended with limited editorial review

Updated Tier Setting description

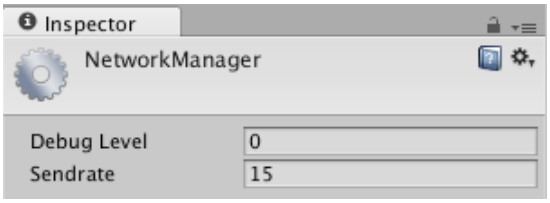
New features added in 5.6

Network Manager

Leave feedback

(This class is part of the old networking system and is deprecated. See NetworkManager for the new networking system).

The **Network Manager** contains two very important properties for making Networked multiplayer games.



The Network Manager

You can access the Network Manager by selecting **Edit > Project Settings > Network** from the menu bar.

Properties

Property:	*Function:*
Debug Level	The level of messages that are printed to the console
Sendrate	Number of times per second that data is sent over the network

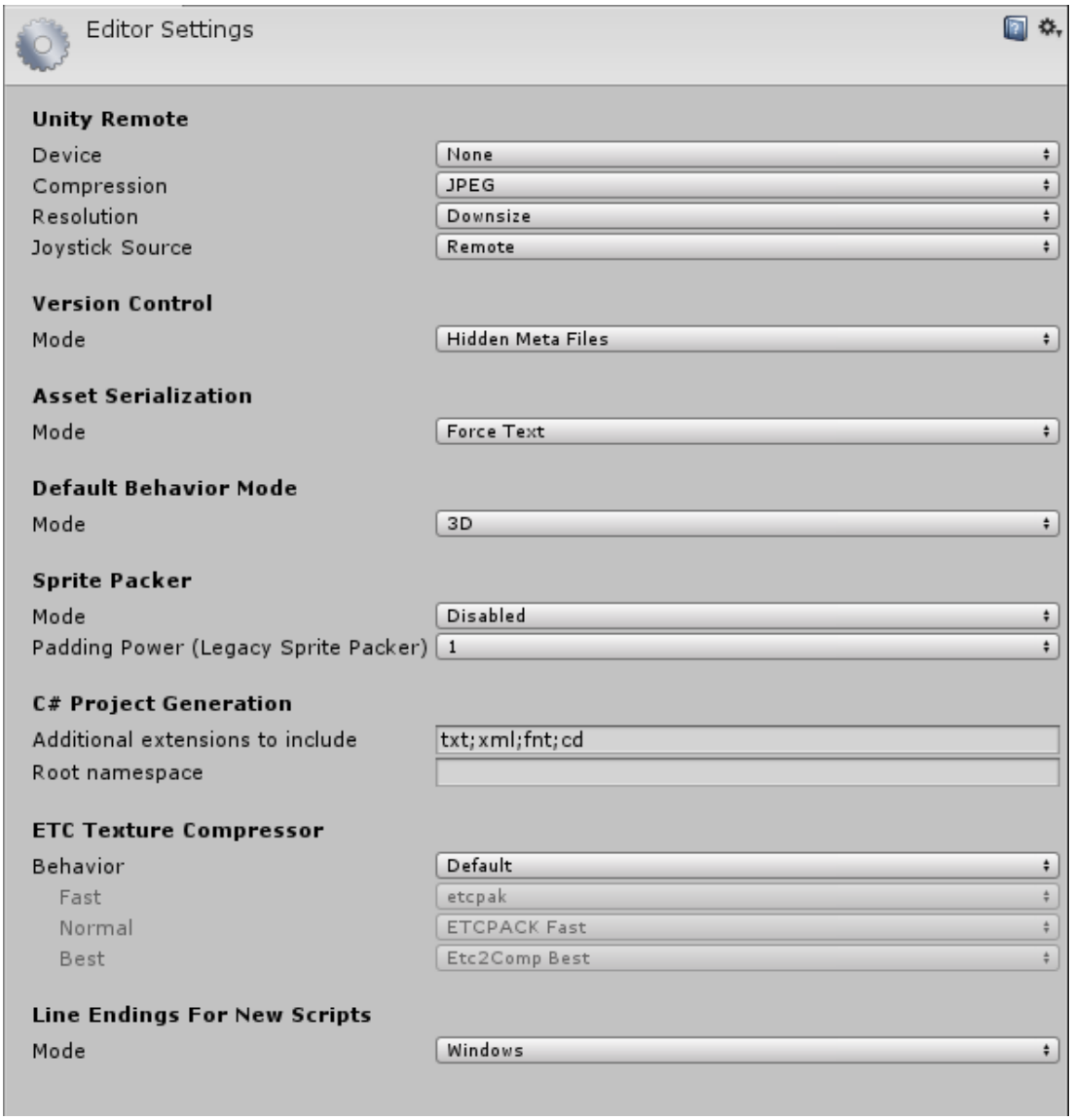
Details

Adjusting the Debug Level can be enormously helpful in fine-tuning or debugging your game's **networking** behaviors. When it is set to **0**, only errors from **networking** will be printed to the console.

The data that is sent at the **Sendrate** intervals (1 second / **Sendrate** = interval) will vary based on the **Network View** properties of each broadcasting object. If the Network View is using **Unreliable**, its data will be send at each interval. If the Network View is using **Reliable Delta Compressed**, Unity will check to see if the Object being watched has changed since the last interval. If it has changed, the data will be sent.

Editor Settings

Leave feedback



Properties

Property:	Function:
Unity Remote	
Device	Unity Remote is a downloadable app designed to help with Android, **iOS** and tvOS development. Use the drop-down to select the device type you want to use for Unity Remote testing.
Compression	Use the drop-down to select the type of image compression Unity uses when transmitting the game screen to the device via Unity Remote. This is set to **JPEG** by default.
JPEG	JPEG usually gives higher compression and performance, but the graphics quality is a little lower. This is the default option.
PNG	PNG gives a more accurate representation of the game display, but can result in lower performance.
Resolution	Use the drop-down to select the resolution the game should run at on Unity Remote. This is set to **Downsize** by default.
Downsize	Choose **Downsize** to display the game at a lower resolution. This results in better performance, but lower graphical accuracy. This is the default option.

Property:	Function:
Normal	Choose **Normal** to display the game at normal resolution. This results in better graphical accuracy, but lower performance.
Joystick Source	Use the drop-down to select the connection source for the joysticks you are using. This is set to **Remote** by default.
Remote	Choose **Remote** to use joysticks that are connected to a device running Unity Remote. This is the default option.
Local	Select **Local** to use joysticks that are connected to the computer running the Editor.
Version Control	
Mode	You can use Unity in conjunction with most common version control tools, including Perforce and PlasticSCM. See documentation on Version Control for more information. Use the drop-down to select the visibility of meta files. See documentation on Version Control for different options available for different systems. This is set to **Hidden Meta Files** by default. For more information on showing or hiding meta files, see the Unity Answers page Visible or hidden meta files.
Hidden Meta Files	Set Unity to hide meta files. This is the default option.
Visible Meta Files	Set Unity to display meta files. This is useful when using version control, because it allows other users and machines to view these meta files.
Asset Serialization	
Mode	Unity uses serialization to load and save Assets and AssetBundles to and from your computer's hard drive. To help with version control merges, Unity can store Scene files in a text-based format (see documentation on text scene format for further details). If Scene merges are not performed, Unity can store Scenes in a more space-efficient binary format, or allow both text and binary Scene files to exist at the same time. Use the drop-down to select which format Unity should use to store serialized Assets. This is set to **Force Text** by default.
Mixed	**Assets** in Binary mode remain in Binary mode, and **assets** in Text mode remain in Text mode. Unity uses Binary mode by default for new **Assets**.
Force Binary	This converts all Assets to Binary mode, and Unity uses Binary mode for new Assets.
Force Text	This converts all Assets to Text mode, and Unity uses Text mode for new Assets. This is the default option.
Default Behavior Mode	
Mode	Unity allows you to choose between 2D or 3D development, and sets up certain default behaviors depending on which one you choose, to make development easier. For more information on the specific behaviors that change with this setting, see documentation on 2D and 3D Mode settings. Use the drop-down to choose a default behaviour for Unity. This is set to **3D** by default.
3D	Choose **3D** to set Unity up for 3D development. This is the default option.
2D	Choose **2D** to set Unity up for 2D development.
Sprite Packer	
Mode	Unity provides a Sprite Packer tool to automate the process of generating atlases from individual **Sprite** Textures. Use these settings to configure the **Sprite** Packer. Use the drop-down to select a **Sprite** Packer Mode. This is set to **Disabled** by default. See documentation on Sprite Atlas for more information.
Disabled	Unity does not pack Atlases. This is the default setting.

Property:	Function:
Enabled For Builds (Legacy Sprite Packer)	Unity packs Atlases for builds only, and not in-Editor Play mode. This **Legacy Sprite Packer** option refers to the Tag-based version of the Sprite Packer, rather than the Sprite Atlas version.
Always Enabled (Legacy Sprite Packer)	Unity packs Atlases for builds and before entering in-Editor Play mode. This **Legacy Sprite Packer** option refers to the Tag-based version rather than the Sprite Atlas version.
Enabled For Builds	Unity packs Atlases for builds only, and not in-Editor Play mode.
Always Enabled	Unity packs Atlases for builds and before entering in-Editor Play mode.
Padding Power (Legacy Sprite Packer)	Use **Padding Power** to set the value that the packing algorithm uses when calculating the amount of space or "padding" to allocate between packed Sprites, and between Sprites and the edges of the generated atlas.
1	This represents the value 2^1. Use this setting to allocate 2 **pixels** between packed Sprites and atlas edges. This is the default setting.
2	This represents the value 2^2. Use this setting to allocate 4 pixels between packed Sprites and atlas edges.
3	This represents the value 2^3. Use this setting to allocate 8 pixels between packed Sprites and atlas edges.
C# Project Generation	
Additional extensions to include	Use this text field to include a list of additional file types to add to the C# Project. Separate each file type with a semicolon. By default, this field contains `txt;xml;fnt;cd`.
Root namespace	Use this text field to fill in the namespace to use for the C# project `RootNamespace` property. See Microsoft developer documentation on Common MSBuild Project Properties for more information. This field is blank by default.
ETC Texture Compressor	
Behavior	Unity allows you to specify the compression tool to use for different compression qualities of ETC Textures. The properties **Fast**, **Normal** and **Best** define the compression quality. These map to the **Compressor Quality** setting in the Texture Importer for the supported platforms. The compression tools available are etcpak, ETCPACK and Etc2Comp. These are all third-party compressor libraries.
Legacy	Select **Legacy** to use the configuration that was available before ETC Texture compression became configurable. This sets the following properties: - **Fast**: ETCPACK Fast - **Normal**: ETCPACK Fast - **Best**: ETCPACK Best
Default	Select **Default** to use the default configuration for Unity. This sets the following properties: - **Fast**: etcpack - **Normal**: ETCPACK Fast - **Best**: Etc2Comp Best

Property:	Function:
Custom	Select **Custom** to customise the **ETC Texture compression** configuration.
Line Endings For New Scripts	
Mode	Use the drop-down to configure file line endings to apply to new **scripts** created within the Editor. Note that configuring these settings does not convert existing **scripts**.
OS Native	Choose **OS Native** to apply line endings based on the OS the Editor is running on.
Unix	Choose **Unix** to apply line endings based on the Unix OS.
Windows	Choose **Windows** to apply line endings based on the Windows OS.

2017–11–03 Documentation updated to reflect multiple changes since 5.6

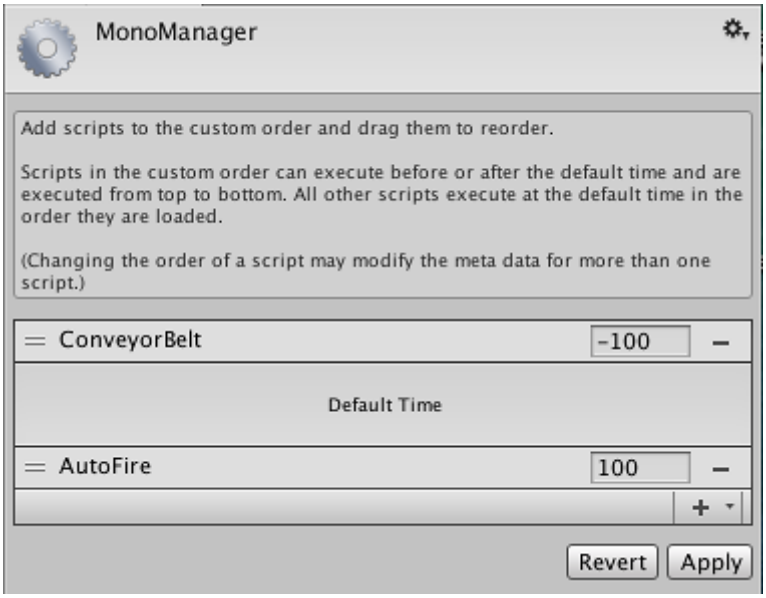
2017–11–03 Page amended with editorial review

Script Execution Order Settings

Leave feedback

See documentation on the execution order of event functions to learn how Unity handle event functions by default.

You can use the **Script Execution Order** settings (menu: **Edit > Project Settings > Script Execution Order**).



Scripts can be added to the **inspector** using the Plus "+" button and dragged to change their relative order. Note that it is possible to drag a script either above or below the **Default Time** bar; those above will execute ahead of the default time while those below will execute after. The ordering of **scripts** in the dialog from top to bottom determines their execution order. All **scripts** not in the dialog execute in the default time slot in arbitrary order.

The numbers shown for each script are the values the scripts are actually ordered by. When a script is dragged to a new position, the number for the script is automatically changed accordingly. When a number is changed, either manually or automatically, it changes the meta file for that script. For this reason it's best if as few of the numbers as possible change when the order is changed. This is why, when possible, only the script that is dragged has its number changed, rather than assigning new numbers to all the scripts.

Preset Manager

Leave feedback

Use Presets to specify default properties for new **components** and **assets**. You cannot set the default properties for Settings Managers.

When you add a component to a **GameObject** or a new asset to your project, Unity uses a default Preset to set the properties of the new item. Default Presets override the Unity factory default settings.

Unity also uses default Presets when you use the **Reset** command in the Component Context Menu in the **Inspector**More info
See in Glossary window.

For Transform components, Unity does not use the position in a default Preset when you add a new GameObject to a scene. In this case, the default position is the middle of the **Scene**More info
See in Glossary view. Use the **Reset** command to apply the position, as well as the rotation and scale, of a default Preset for a Transform.

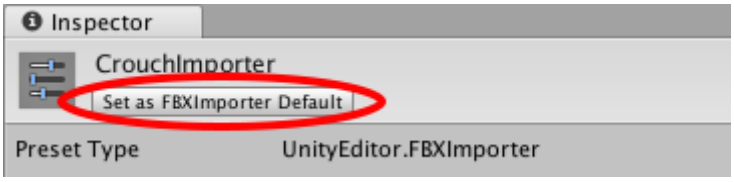
Specifying a Preset to use for default settings

You can specify a Preset to use for default settings with the **Inspector** window, the **Preset Manager**, or by dragging and dropping.

To specify default settings with the **Inspector** window:

Select a Preset in the **Project** window.

In the **Inspector** window, click **Set as Default**.



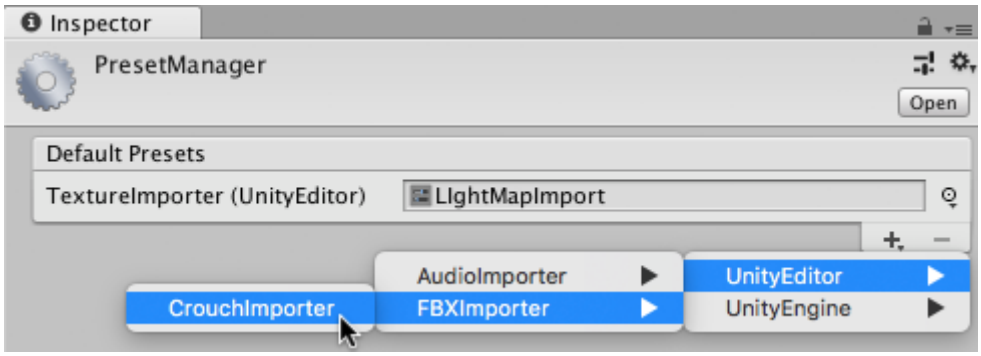
Set as Default (red) in the **Inspector** window

To specify default settings with the **Preset Manager**:

If you don't already have a Preset in your project to use for default settings, create one.

Open the Preset Manager by choosing **Edit > Project Settings > Preset Manager**.

Click **+** and select the item that you want to use with a default Preset.
A default Preset for the selected item appears in the **Preset Manager** list.



For example, click **+** and select CrouchImporter to specify it as the default for imported models

Drag and drop a Preset from the **Project** window to the **Preset Manager** to add a new default Preset or change an existing default Preset.

Changing and removing default Presets

Use **Preset Manager** to change a default Preset. You can also drag and drop a Preset from the **Project** window to the **Preset Manager** to change an existing default Preset.

There are two ways to remove a default Preset: from **Preset Manager** or from the **Inspector** window.

To change a default Preset in **Preset Manager**:

Open the Preset Manager by choosing **Edit > Project Settings > Preset Manager**.
Click the drop-down menu next to the default Preset of an object type to select a Preset.
The selected Preset becomes the new default Preset.

To remove a default Preset in the **Inspector**:

Select a Preset in the Project window.
In the Inspector window, click **Remove From**.

To remove a default Preset in **Preset Manager**:

Open the Preset Manager by choosing **Edit > Project Settings > Preset Manager**.
Select the default Preset that you want to remove from the list of default Presets.
Click **-** to remove the selected default Preset.

2017–03–27 Page published with limited editorial review New feature in 2018.1

Visual Studio C# integration

Leave feedback

Benefits of using Visual Studio

A more sophisticated C# development environment. Think smart autocompletion, computer-assisted changes to source files, smart syntax highlighting and more.

The difference between Community, Express and Pro

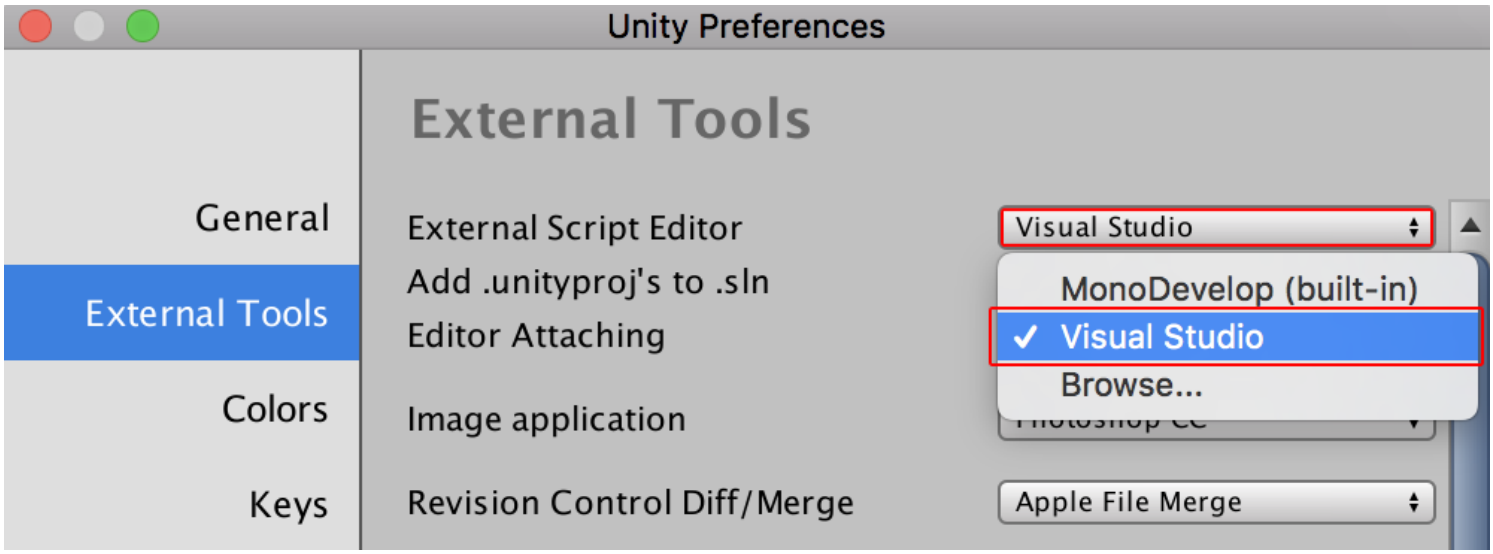
VisualStudio C# is an Integrated Development Environment (IDE) tool from Microsoft. Visual Studio now comes in three editions, Community (free to use) Professional (paid) and Enterprise (paid). A comparison of feature differences between versions is available on the Visual Studio website.

Unity's Visual Studio integration allows you to create and maintain Visual Studio project files automatically. Also, VisualStudio will open when you double click on a script or on an error message in the Unity console.

Using Visual Studio with Unity

Follow these steps to configure the Unity Editor to use Visual Studio as its default IDE:

In Unity, go to **Edit > Preferences**, and make sure that Visual Studio is selected as your preferred external editor.



External Tool Settings

Next, doubleclick a C# file in your project. Visual Studio should automatically open that file for you.

You can edit the file, save, and switch back to Unity to test your changes.

A few things to watch out for

Even though Visual Studio comes with its own C# compiler, and you can use it to check if you have errors in your c# **scripts**, Unity still uses its own C# compiler to compile your **scripts**. Using the Visual Studio compiler is still quite useful, because it means you don't have to switch to Unity all the time to check if you have any errors or not.

Visual Studio's C# compiler has some more features than Unity's C# compiler currently supports. This means that some code (especially newer c# features) will not throw an error in Visual Studio but will in Unity.

Unity automatically creates and maintains a Visual Studio .sln and .csproj file. Whenever somebody adds/renames/moves/deletes a file from within Unity, Unity regenerates the .sln and .csproj files. You can add files to your solution from Visual Studio as well. Unity will then import those new files, and the next time Unity creates the project files again, it will create them with this new file included.

RenderDoc Integration

Leave feedback

The Editor supports integrated launching and capture of the RenderDoc graphics debugger, for detailed frame introspection and debugging.

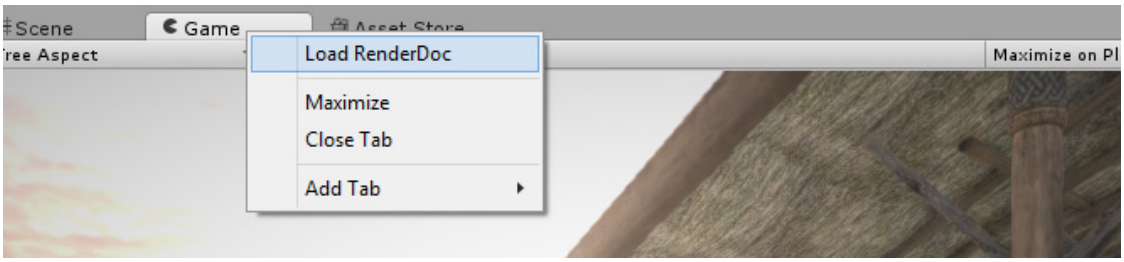
The integration is only supported for RenderDoc versions 0.26 or later, so if an earlier version is currently installed it is required that you update to at least version 0.26.

Note: While the integration is only available in the Editor, it is quite possible to use RenderDoc as normal with no extra setup in standalone player builds.

Note: Frames can only be captured if Unity is running on a platform and API that RenderDoc supports - at time of writing that means Windows only, and either DirectX 11 or **OpenGL Core** profile. If another API is in use, the RenderDoc integration will be temporarily disabled until a supported API is enabled.

Loading RenderDoc

If a RenderDoc installation is detected, then at any time after loading the Editor you can right click on the tab for the Game View or Scene View and click the 'Load RenderDoc' option. This will reload the graphics device so you must save any changes, but afterwards RenderDoc will be ready to capture without having to restart the editor or build a standalone player.

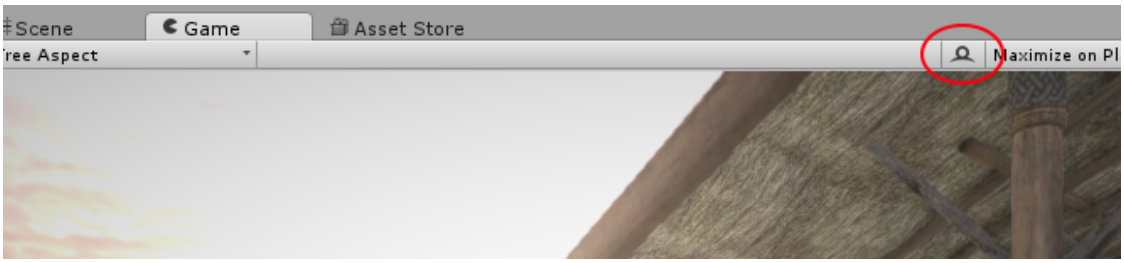


Loading RenderDoc at runtime

Note: You can also launch the Editor via RenderDoc as normal, or pass the -load-renderdoc command line option to load RenderDoc from startup.

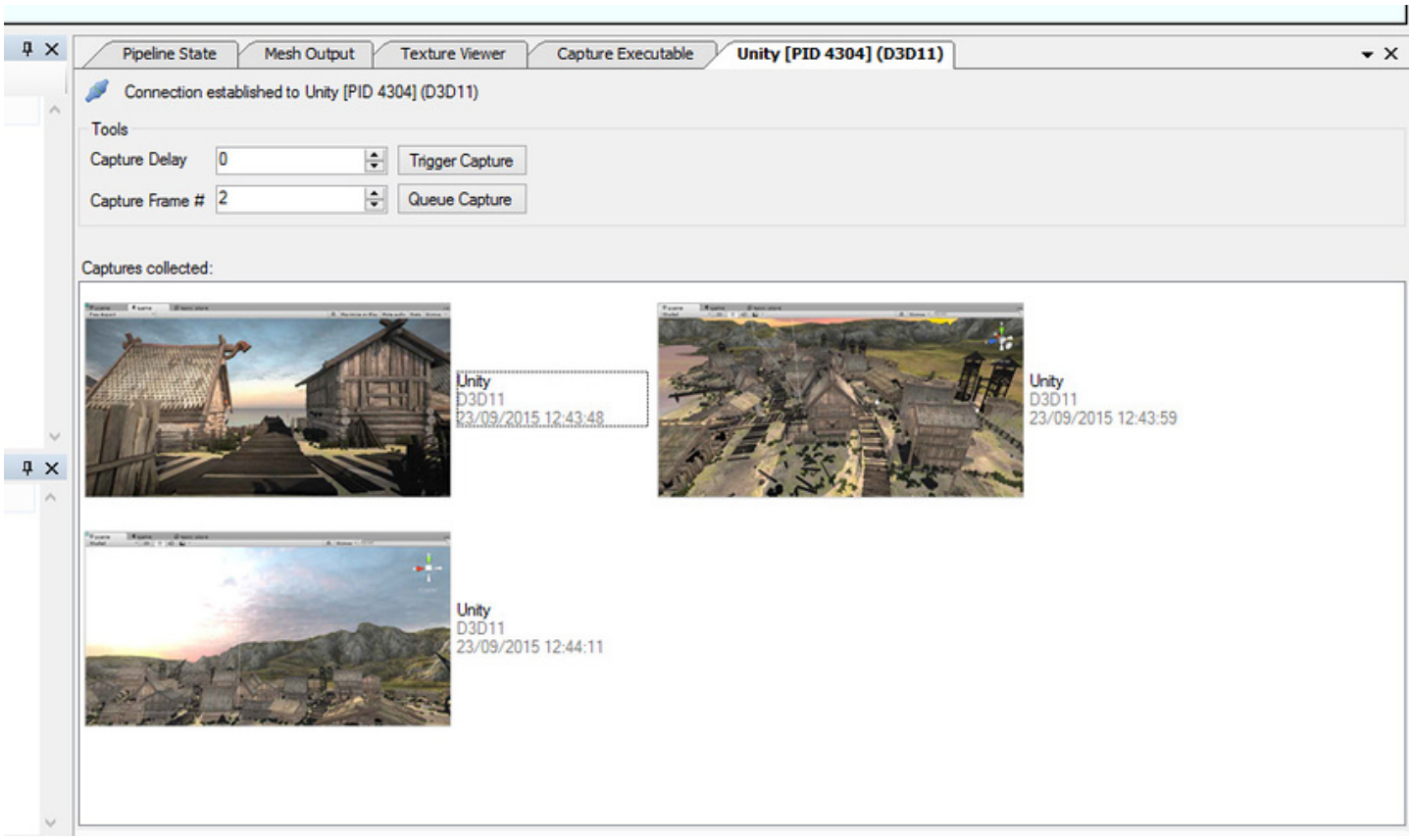
Capturing a frame with RenderDoc

When a compatible version of RenderDoc is detected as loaded into the Editor, a new button will appear on the right side of the **toolbar** on the Game and **Scene** Views.



Capturing a frame with RenderDoc

Pressing this button will trigger a capture of the next frame of **rendering** for the view. If the RenderDoc tool **UI** has not been opened, a new instance will be launched to show the capture, and if it is already running the newest capture will automatically appear there. From there you can open the capture and debug using the tool.



List of frame captures in RenderDoc

Including shader debug information

By default to optimise the size of DirectX11 shaders, debugging information is stripped out. This means that constants and resources will have no names, and the **shader** source will not be available. To include this debugging information in your **shader**, include **#pragma enable_d3d11_debug_symbols** in your **shader**'s **CGPROGRAM** block.

Alternative graphics debugging techniques

If you build a standalone player using D3D11, you can capture a frame and debug using the Visual Studio graphics debugger.

Editor Analytics

Leave feedback

The Unity editor is configured to send anonymous usage data back to Unity. This information is used to help improve the features of the editor. The **analytics** are collected using Google **Analytics**. Unity makes calls to a URI hosted by Google. The URN part of the URI contains details that describe what editor features or events have been used.

Examples of collected data

The following are examples of data that Unity might collect.

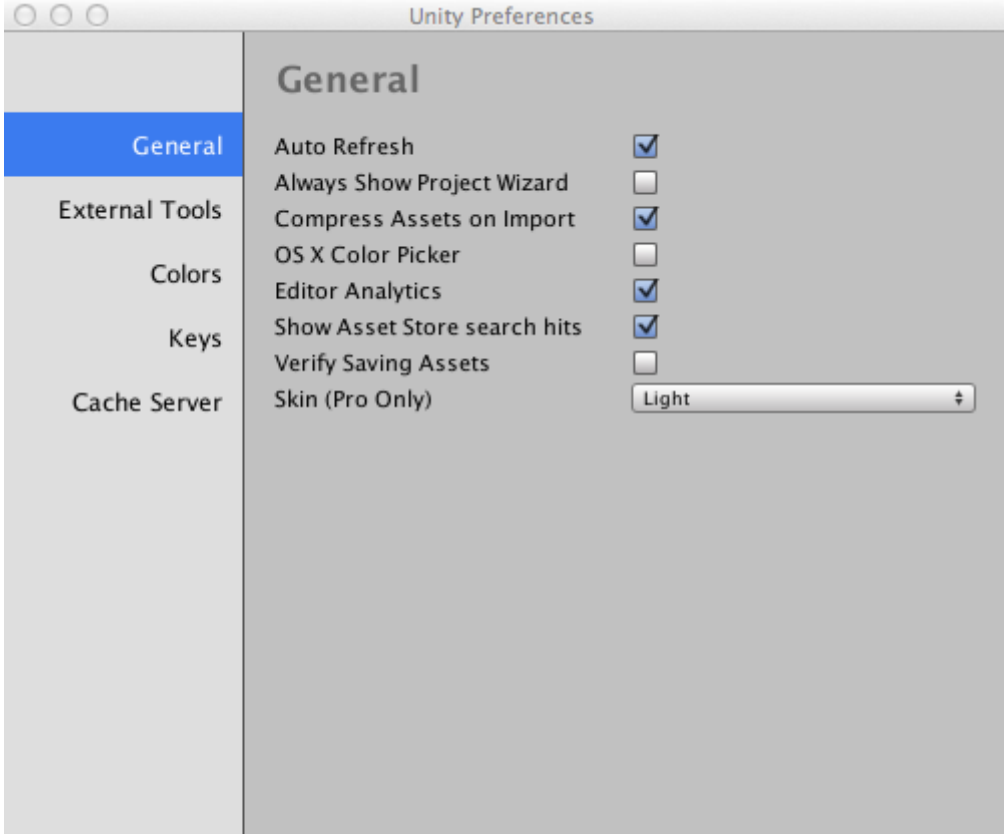
Which menu items have been used. If some menu items are used rarely or not at all we could in the future simplify the menuing system.

Build times. By collecting how long builds take to make we can focus engineering effort on optimizing the correct code.

Lightmap baking. Again, timing and reporting how long it takes for light maps to bake can help us decide how much effort to spend on optimizing this area.

Disabling Editor Analytics

If you do not want to send anonymous data to Unity then the sending of Editor Analytics can be disabled. To do this untick the box in the Unity Preferences General tab.



Editor analytics in the preferences pane.

Check For Updates

Leave feedback

Unity checks whether updates are available. This check happens either when Unity is started, or when you choose the Help->Check for Updates menu item. The update check sends the current Unity revision number (the five digit number that appears in brackets after the version name in the About Unity dialog) to the update server where is it compared with the most-up-to-date released version. If a newer version of Unity is available the following dialog is shown:

Window displayed when there is a newer version of Unity available for download.

If the version in use is the most up-to-date then the following dialog is shown:

Window displayed when Unity is updated to the latest version.
Click the Download new version button to be taken to the website where you can download the new version.

Update Check Frequency

The response from the server also contains a time interval which suggests when the next update check should be made. This allows the update check to be made less frequently when Unity is not expecting updates to be made available.

Skipping Update Versions

If you are in the middle of a project you may not want to update to a new version of Unity. Ticking the Skip this version button on the Unity Editor Update Check dialog will prevent Unity from telling you about this update.

Disabling the Update Check

It is not possible to disable the check for updates. The Check For Updates tick box on the dialog controls whether you are notified of updates (if they are available) when Unity starts. Even if you have unticked the Check for Updates option you can still check for updates by using the Help->Check for Updates menu item.

2017–11–10 Page amended with no editorial review

IME in Unity

Leave feedback

What is Input Method Editor (IME)?

An input method is an operating system **component** or program that allows users to enter characters and symbols not found on their input device. For instance, on the computer, this allows the user of 'Western' keyboards to input Chinese, Japanese, Korean and Indic characters. On many hand-held devices, such as mobile phones, it enables using the numeric keypad to enter Latin alphabet characters.

The term input method generally refers to a particular way to use the keyboard to input a particular language, for example the Cangjie method, the pinyin method, or the use of dead keys.

IME and Unity

Unity provides IME support, which means that you can write non-ASCII characters in all your graphical user interfaces. This Input Method is fully integrated with the engine so you do not need to do anything to activate it. In order to test it, just change your keyboard language to a non-ASCII language (e.g. Japanese) and just start writing your interface.

For more info and optimization when writting non-ASCII characters, check the **character** option in the font properties.

iOS

This feature is not supported on **iOS** devices yet.

Android

This feature is not supported on Android devices yet.

Special folder names

Leave feedback

You can usually choose any name you like for the folders you create to organise your Unity project. However, there are a number of folder names that Unity interprets as an instruction that the folder's contents should be treated in a special way. For example, you must place Editor **scripts** in a folder called **Editor** for them to work correctly.

This page contains the full list of special folder names used by Unity.

Assets

The **Assets** folder is the main folder that contains the Assets used by a Unity project. The contents of the Project window in the Editor correspond directly to the contents of the Assets folder. Most API functions assume that everything is located in the Assets folder, and so don't require it to be mentioned explicitly. However, some functions do need to have the Assets folder included as part of a pathname (for example, certain functions in the AssetDatabase class).

Editor

Scripts placed in a folder called **Editor** are treated as Editor scripts rather than runtime scripts. These scripts add functionality to the Editor during development, and are not available in builds at runtime.

You can have multiple Editor folders placed anywhere inside the Assets folder. Place your Editor scripts inside an Editor folder or a subfolder within it.

The exact location of an Editor folder affects the time at which its scripts will be compiled relative to other scripts (see documentation on Special Folders and Script Compilation Order for a full description of this). Use the EditorGUIUtility.Load function in Editor scripts to load Assets from a Resources folder within an Editor folder. These Assets can only be loaded via Editor scripts, and are stripped from builds.

Note: Unity does not allow **components** derived from MonoBehaviour to be assigned to **GameObjects** if the scripts are in the Editor folder.

Editor Default Resources

Editor scripts can make use of Asset files loaded on-demand using the EditorGUIUtility.Load function. This function looks for the Asset files in a folder called **Editor Default Resources**.

You can only have one Editor Default Resources folder and it must be placed in the root of the Project; directly within the Assets folder. Place the needed Asset files in this Editor Default Resources folder or a subfolder within it. Always include the subfolder path in the path passed to the EditorGUIUtility.Load function if your Asset files are in subfolders.

Gizmos

Gizmos allow you to add graphics to the **Scene View** to help visualise design details that are otherwise invisible. The Gizmos.DrawIcon function places an icon in the **Scene** to act as a marker for a special object or position. You must place the image file used to draw this icon in a folder called **Gizmos** in order for it to be located by the DrawIcon function.

You can only have one Gizmos folder and it must be placed in the root of the Project; directly within the Assets folder. Place the needed Asset files in this Gizmos folder or a subfolder within it. Always include the subfolder path in the path passed to the Gizmos.DrawIcon function if your Asset files are in subfolders.

Plug-ins

You can add plug-ins to your Project to extend Unity's features. Plug-ins are native DLLs that are typically written in C/C++. They can access third-party code libraries, system calls and other Unity built-in functionality. Always place plug-ins in a folder called **Plugins** for them to be detected by Unity.

You can only have one Plugins folder and it must be placed in the root of the Project; directly within the Assets folder.

See Special folders and script compilation order for more information on how this folder affects script compilation, and Plugin Inspector for more information on managing plug-ins for different target platforms.

Resources

You can load Assets on-demand from a script instead of creating instances of Assets in a Scene for use in gameplay. You do this by placing the Assets in a folder called **Resources**. Load these Assets by using the Resources.Load function.

You can have multiple Resources folders placed anywhere inside the Assets folder. Place the needed Asset files in a Resources folder or a subfolder within it. Always include the subfolder path in the path passed to the Resources.Load function if your Asset files are in subfolders.

Note that if the Resources folder is an Editor subfolder, the Assets in it are loadable from Editor scripts but are stripped from builds.

Standard Assets

When you import a Standard Asset package (menu: **Assets** > **Import Package**) the Assets are placed in a folder called **Standard Assets**. As well as containing the Assets, these folders also have an effect on script compilation order; see the page on Special Folders and Script Compilation Order for further details.

You can only have one Standard Assets folder and it must be placed in the root of the Project; directly within the Assets folder. Place the needed Assets files in this Standard Assets folder or a subfolder within it.

StreamingAssets

You may want the Asset to be available as a separate file in its original format although it is more common to directly incorporate Assets into a build. For example, you need to access a video file from the filesystem rather than use it as a MovieTexture to play that video on **iOS**. Place a file in a folder called **StreamingAssets**, so it is copied unchanged to the target machine, where it is then available from a specific folder. See the page about Streaming Assets for further details.

You can only have one StreamingAssets folder and it must be placed in the root of the Project; directly within the Assets folder. Place the needed Assets files in this StreamingAssets folder or a subfolder within it. Always include the subfolder path in the path used to reference the streaming asset if your Asset files are in subfolders.

Hidden Assets

During the import process, Unity completely ignores the following files and folders in the **Assets** folder (or a sub-folder within it):

Hidden folders.
Files and folders which start with '**.**'.
Files and folders which end with '**~**'.
Files and folders named **cvs**.
Files with the extension **.tmp**.
This is used to prevent importing special and temporary files created by the operating system or other applications.

Exporting Packages

Leave feedback

As you build your game, Unity stores a lot of metadata about your **assets**, such as import settings and links to other **assets**, among other information. If you want to transfer your **assets** into a different project and preserve all this information, you can export your **assets** as a **Custom Package**.

See Packages for detailed information on using **Asset packages**, including importing and exporting.

Exporting New Packages

Use **Export Package** to create your own **Custom Package**.

Open the project you want to export assets from.
Choose **Assets > Export Package...** from the menu to bring up the **Exporting Package** dialog box. (See *Fig 1: Exporting Package dialog box*.)
In the dialog box, select the assets you want to include in the package by clicking on the boxes so they are checked.
Leave the **include dependencies** box checked to auto-select any assets used by the ones you have selected.
Click on **Export** to bring up File Explorer (Windows) or Finder (Mac) and choose where you want to store your package file. Name and save the package anywhere you like.
HINT: When exporting a package Unity can export all dependencies as well. So, for example, if you select a **Scene** and export a package with all dependencies, then all models, textures and other assets that appear in the scene will be exported as well. This can be a quick way of exporting a bunch of assets without manually locating them all.

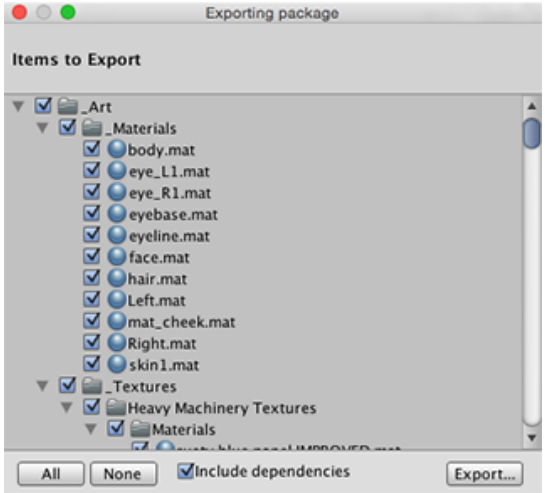


Fig 1: Exporting Package dialog box

Exporting Updated Packages

Sometimes you may want to change the contents of a package and create a newer, updated version of your asset package. To do this:

Select the asset files you want in your package (select both the unchanged ones and the new ones).

Export the files as described above in ***Export Package*** , above.

NOTE: You can re-name an updated package and Unity will recognise it as an update, so you can use incremental naming, for example: MyAssetPackageVer1, MyAssetPackageVer2.

HINT: It is not good practise to remove files from packages and then replace them with the same name: Unity will recognise them as different and possibly conflicting files and so display a warning symbol when they are imported. If you have removed a file and then decide to replace it, it is better to give it a different, but related name to the original.

2018–04–25 Page amended with limited editorial review

Version Control

Leave feedback

You can use Unity in conjunction with most common **version control** tools, including **Perforce** and PlasticSCM. This section gives details of the tools and options available and how to work with them.

Version control integration

Leave feedback

Unity supports version control integration with Perforce and Plastic SCM. Refer to those pages for specific information regarding your choice of version control.

Using a **version control system** makes it easier for a user (or multiple users) to manage their code. It is a repository of files with monitored access. In the case of Unity, this is all the files associated with a Unity project. With version control, it is possible to follow every change to the source, along with information on who made the change, why they made it, and what they changed. This makes it easy to revert back to an earlier version of the code, or to compare differences in versions. It also becomes easier to find out when a bug first occurred, and with what changes might have caused it.

Setting up your version control in Unity

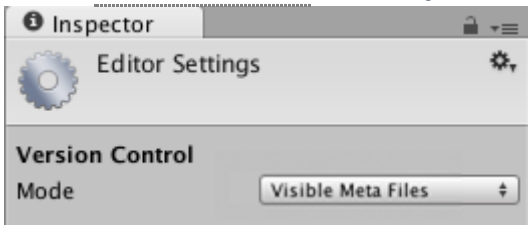
Get your version control software setup according to its instructions, then follow these steps:

Set up or sync a workspace on your computer using your chosen client (refer to the Plastic SCM Integration guide or the Perforce Integration guide for help with this step).

Copy an existing project into the workspace, or start Unity and create a new project in the workspace.

Open the project and go to **Edit** > **Project Settings** > **Editor**.

Under **Version Control**, choose your **Mode** according to the version control system that you chose.



Fill out your version control settings, such as username / password / server / workspace.

Keep **Automatic add** checked if you want your files to be automatically added to version control when they're added to the project (or the folder on disk). Otherwise, you need to add new files manually.

You have the option to work offline. This mode is only recommended if you know how to manually integrate changes back into your version control software (Working offline with Perforce).

You can edit the **Asset Serialization**, **Default Behaviour Mode** and **Sprite Packer** options to suit your team's preferences and choice of version control.

Click **Connect** and verify that "Connected" is displayed above the **Connect** button after a short while.

Use your standard client (e.g. p4v) to make sure that all files in the *Assets* and *ProjectSettings* folders (including files ending with *.meta*) are added.

N.B. At any point you can go to the **Prefences** menu and select **External Tools** and adjust your Revision Control Diff/Merge tool.

Using version control

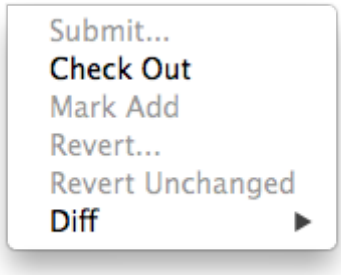
At this point you should be able to do most of the important version control operations directly by right-clicking on the **assets** in the **project view**, instead of going through the version control client. The version control operations vary depending on which version control you choose, this table shows what actions are directly available for each version control:

Version Control Operation	Description	Perforce	Plastic SCM
Check Out	Allows changes to be made to the file	Yes	Yes
Diff against head	Compares differences between file locally and in the head	Yes	Yes
Get Latest	Pull the latest changes and update file	Yes	No*
Lock	Prevents other users from making changes to file	Yes	No**
Mark Add	Add locally but not into version control	Yes	Yes
Resolve Conflicts	To resolve conflicts on a file that has been changed by multiple users	Yes	No***
Revert	Discards changes made to open changed files	Yes	Yes
Revert Unchanged	Discards changes made to open unchanged files	Yes	Yes
Submit	Submits current state of file to version control	Yes	Yes
Unlock	Releases lock and allows changes to be made by anyone	Yes	No**

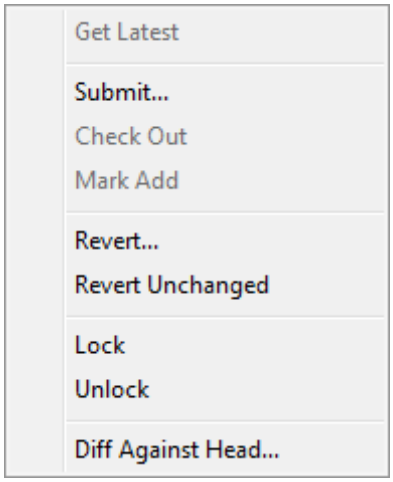
* To get the latest changes and update the file using Plastic SCM, you need to use the version control window.

** Locking and Unlocking using Plastic SCM require you to edit a specific Plastic SCM lock file externally, see the Plastic SCM Integration page for more information.

*** Conflicts are shown within the version control menu but resolved in the Plastic SCM GUI.

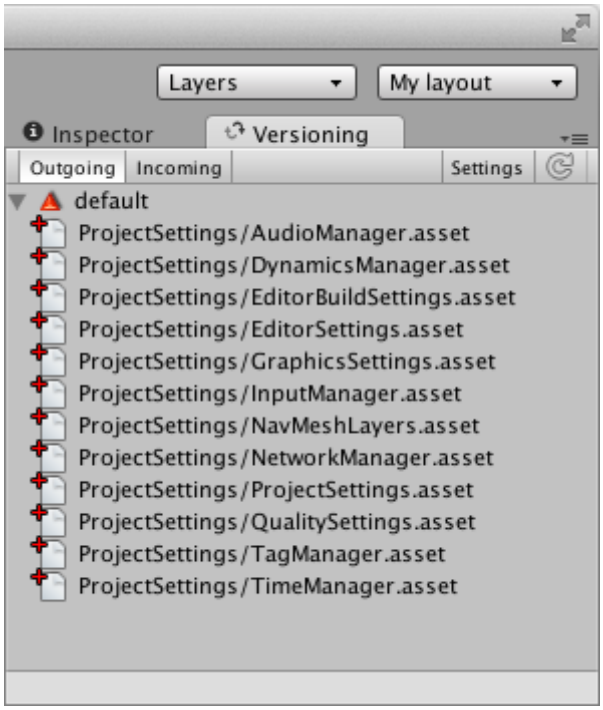


Plastic SCM Version Control Operations on Mac



Version Control Window

You can overview the files in your changelist from the **Version Control Window** (**Window** > **Asset Management** > **Version Control**). It is shown here docked next to Inspector in the editor:



The 'Outgoing' tab lists all of the local changes that are pending a commit into version control whereas the 'Incoming' tab lists all of the changes that need to be pulled from version control.

By right clicking assets or changelists in this window you perform operations on them. To move assets between changelists just drag the assets from one changelist to the header of the target changelist.

Icons

The following icons are displayed in Unity editor to visualize version control status for files/assets:

Icon	Meaning	Additional information
	File added locally	Pending add into version control
	File added to version control by another user	Pending add into version control
	File is checked out by you	Checked out locally
	File is checked out by another user	Checked out remotely
	There has been a conflict merging this file	Needs to be resolved
	File has been deleted by you	Pending deletion in version control
	File has been deleted by another user	Pending deletion in version control
	File is not yet under version control	n/a
	File is locked by you	Cannot be modified by other users
	File is locked by another user	Cannot be modified by you
	Another user has checked in a new version of this file	Use "Apply Incoming Changes" to get latest version

The server is requesting the version control status of this file, or is waiting for a response	You can only see this when using a centralised version control system like Perforce

Things to note:

Certain version controls will not allow you to edit assets until they're marked as *Checked out* (unless you have **Work offline** checked).
When saving changes to a .**scene** file it will automatically be checked out.
Project Settings **inspectors** have a **checkout** button in the bottom right that allow you to checkout settings.
A yellow warning will often appear to remind you to check out items in order to make changes to them, this mostly applies to Project Settings inspectors.
In Plastic SCM automatically generated assets such as light maps are automatically added/checked out.

Automatic revert of unchanged files on submit

When working with assets, Unity automatically checks out both the asset file as well as the associated .meta file. In most situations however, the .meta file is not modified, which might cause some extra work e.g. when merging branches at a later point.

Offline mode

Unity supports working in offline mode, e.g. to continue working without a network connection to your version control repository.

Select **Work offline** from Version Control Settings if you want to be able to work disconnected from version control.

Allow Async Update

Unity supports asynchronous version control status queries for some version control providers, such as Perforce. This option allows those providers to update the version control status of files without stalling Unity. Use this option when the connection to your version control server has high latency.

If you experience stalls during status queries, go to **Version Control Settings** and select **Allow Async Update**.
Note: Only status queries are asynchronous. Operations that change the state of files, or require up-to-date knowledge of file status, are performed synchronously.

Troubleshooting

If Unity cannot commit your changes to your version control client, (for example, if the server is down or license issues occur), your changes are stored in a separate changeset.

Working with the Asset Server

To learn more about working with the Asset Server (Unity's internal version control system), see documentation on the Asset Server.

Working with other version control systems

To work with a version control system unsupported by Unity, select **MetaData** as the **Mode** for Version Control in the Editor Settings. This allows you to manage the source Assets and metadata for those Assets with a version control system of your choice. For more on this, see documentation on External Version Control Systems.

2017–02–12 Page amended with limited editorial review

Documentation on asynchronous version control status queries added in Unity 2017.3.

Perforce Integration

Leave feedback

For more information on **Perforce** you can visit www.perforce.com.

Setting up Perforce

Refer to perforce documentation if you encounter any problems with the setup process on the version control page.

Working Offline with Perforce

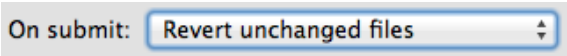
Only use this if you know how to work offline in Perforce without a Sandbox. Refer to the Perforce documentation for further information.

Troubleshooting

If Unity for some reason cannot commit your changes to Perforce, e.g. if server is down, license issues **etc**., your changes will be stored in a separate changeset. If the console doesn't list any info about the issue you can use the P4V client for Perforce to submit this changeset to see the exact error message.

Automatic revert of unchanged files on submit

It's possible to configure Perforce to revert unchanged files on submit, which is done in P4V by selecting **Connection->Edit Current Workspace...,** viewing the Advanced tab and setting the value of On submit to **Revert unchanged files**:



Plastic SCM Integration

Leave feedback

For more information on Plastic SCM you can visit their website.

Setting up Plastic SCM

Refer to the Plastic SCM documentation if you encounter any problems with the set up process on the version control page.

Checking Files Out with Plastic SCM

Plastic SCM automatically checks files out if they have been modified, this makes it more convenient for you. The only files that require specific checking out instructions are **Project Settings** files otherwise you can't change them.

Resolving Conflicts and Merging with Plastic SCM

A merge is likely to happen when you have edited something in your project locally which has also been edited remotely (a conflict). This means you will need to review the changes before the merge can be performed. If Unity recognises that a merge must be completed before changes can be submitted then you will be prompted by Unity to complete the merge, this will take you to the Plastic SCM client.

If incoming changes conflict with local changes then a question mark icon will appear on conflicting files in the incoming changes window. Here is a quick guide to resolving conflicts and merging with Plastic SCM:

In the **Version Control** window click on the'Apply all incoming changes' button, this will automatically take you to the Plastic SCM GUI client.
Within the client window you will be able to click 'Explain merge' for a more visual understanding of the changes. Now click 'Process all merges' and another window will display.
Here you will be shown the individual conflicts and given the option to choose what changes you want to keep or discard.
Once you have solved the conflicts click on save and exit, this will have completed the merge operation.
You now have to push the changes like normal through Unity's version control window.

Locking Files with Plastic SCM

In order to lock files using Plastic SCM there are a few steps to follow:

The first thing you must do is create a lock.conf file and make sure it is placed within your server directory. You can find your server directory from "../PlasticSCM/server".

In your lock.conf file you must specify the repository you are working on and the server that will complete the lock checks. Here is an example:

```
rep:default lockserver:localhost:8087
*.unity
*.unity.meta
```

In this case all .unity and .unity.meta files are going to be locked for checkout on repository 'default'.

You may want to restart your server at this point, you can do this by opening a terminal/command line window and locating the server directory. Once in the directory you can restart the server by typing:

```
./plasticsd restart
```

Now go back to Unity and check out a file that you expect to be locked, then go back to the terminal/command line and type:

```
cm listlocks
```

If the steps have been followed correctly the terminal/command line window should now display a list of locked files. You can also test if this has worked by trying to check out the same file using a different user, an error will appear in Unity's console saying the file is already checked out by another user.

For more information you can visit the Plastic SCM lock file documentation.

Distributed and offline work with Plastic SCM

To find more about working in distributed mode (DVCS) and offline with Plastic SCM check the Distributed Version Control Guide.

Using external version control systems with Unity

Leave feedback

Unity offers an Asset Server add-on product for easy integrated versioning of your projects and you can also use **Perforce** and **PlasticSCM** as external tools (see Version Control Integration for further details). If you for some reason are not able use these systems, it is possible to store your project in any other **version control** system, such as Subversion or Bazaar. This requires some initial manual setup of your project.

Before checking your project in, you have to tell Unity to modify the project structure slightly to make it compatible with storing assets in an external version control system. This is done by selecting **Edit->Project Settings->Editor** in the application menu and enabling External Version Control support by selecting **Visible Meta Files** in the dropdown for Version Control. This will show a text file for every asset in the `Assets` directory containing the necessary bookkeeping information required by Unity. The files will have a `.meta` file extension with the first part being the full file name of the asset it is associated with. Moving and renaming assets within Unity should also update the relevant `.meta` files. However, if you move or rename assets from an external tool, make sure to syncronize the relevant `.meta` files as well.

When checking the project into a version control system, you should add the `Assets`, `Packages` and the `ProjectSettings` directories to the system. The `Library` directory should be completely ignored - when using .meta files, it's only a local cache of imported **assets**.

When creating new assets, make sure both the asset itself and the associated `.meta` file is added to version control.

Example: Creating a new project and importing it to a Subversion repository.

First, let's assume that we have a subversion repository at `svn://my.svn.server.com/` and want to create a project at `svn://my.svn.server.com/MyUnityProject`. Then follow these steps to create the initial import in the system:

Create a new project inside Unity and call it `InitialUnityProject`. You can add any initial assets here or add them later on.
Enable **Visible Meta files** in **Edit->Project Settings->Editor**
Quit Unity (this ensures that all the files are saved).
Delete the `Library` directory inside your project directory.
Import the project directory into Subversion. If you are using the command line client, this is done like this from the directory where your initial project is located: `svn import -m"Initial project import" InitialUnityProject svn://my.svn.server.com/MyUnityProject` If successful, the project should now be imported into subversion and you can delete the `InitialUnityProject` directory if you wish.
Check out the project back from subversion `svn co svn://my.svn.server.com/MyUnityProject` and check that the `Assets`, `Packages` and `ProjectSettings` directory are versioned.
Open the checked out project with Unity by launching it while holding down the **Option** or the left **Alt** key. Opening the project will recreate the `Library` directory in step 4 above.
Optional: Set up an ignore filter for the unversioned `Library` directory: `svn propedit svn:ignore MyUnityProject/` Subversion will open a text editor. Add the Library directory.
Finally, commit the changes. The project should now be set up and ready: `svn ci -m"Finishing project import" MyUnityProject`

Smart Merge

Leave feedback

Unity incorporates a tool called **UnityYAMLMerge** that can merge **scene** and **prefab** files in a semantically correct way. The tool can be accessed from the command line and is also available to third party **version control** software.

Setting Up Smart Merging in Unity

In the **Editor Settings** (menu: **Edit > Project Settings > Editor**), you have the option to select a third party version control tool (Perforce or PlasticSCM, for example). When one of these tools is enabled, you will see a *Smart Merge* menu under the *Version Control* heading. The menu has four options:

Off: use only the default merge tool set in the preferences with no smart merging.
Premerge: enable smart merging, accept clean merges. Unclean merges will create premerged versions of base, theirs and mine versions of the file. Then, use these with the default merge tool.
Ask: enable smart merging but when a conflict occurs, show a dialog to let the user resolve it (this is the default setting).

Setting up UnityYAMLMerge for Use with Third Party Tools

The UnityYAMLMerge tool is shipped with the Unity editor; assuming Unity is installed in the standard location, the path to UnityYAMLMerge will be:

```
C:\Program Files\Unity\Editor\Data\Tools\UnityYAMLMerge.exe

or

C:\Program Files (x86)\Unity\Editor\Data\Tools\UnityYAMLMerge.exe
```

...on Windows and

```
/Applications/Unity/Unity.app/Contents/Tools/UnityYAMLMerge
```

...on Mac OSX (use the *Show Package Contents* command from the Finder to access this folder).

UnityYAMLMerge is shipped with a default fallback file (called mergespecfile.txt, also in the Tools folder) that specifies how it should proceed with unresolved conflicts or unknown files. This also allows you to use it as the main merge tool for version control systems (such as git) that don't automatically select merge tools based on file extensions. The most common tools are already listed by default in mergespecfile.txt but you can edit this file to add new tools or change options.

You can run UnityYAMLMerge as a standalone tool from the command line (you can see full usage instructions by running it without any arguments). Set-up instructions for common version control systems are given below.

P4V

Go to Preferences > Merge.
Select *Other application*.
Click the *Add* button.
In the extension field, type `.unity`.
In the Application field, type the path to the UnityYAMLMerge tool (see above).
In the Arguments field, type `merge -p %b %1 %2 %r`

Click Save.
Then, follow the same procedure to add the `.prefab` extension.

Git

Add the following text to your `.git` or `.gitconfig` file:

```
[merge]
tool = unityyamlmerge

[mergetool "unityyamlmerge"]
trustExitCode = false
cmd = '<path to UnityYAMLMerge>' merge -p "$BASE" "$REMOTE" "$LOCAL" "$MERGED"
```

Mercurial

Add the following text to your `.hgrc` file:

```
[merge-patterns]
**.unity = unityyamlmerge
**.prefab = unityyamlmerge

[merge-tools]
unityyamlmerge.executable = <path to UnityYAMLMerge>
unityyamlmerge.args = merge -p --force $base $other $local $output
unityyamlmerge.checkprompt = True
unityyamlmerge.premerge = False
unityyamlmerge.binary = False
```

SVN

Add the following to your `~/.subversion/config` file:

```
[helpers]
merge-tool-cmd = <path to UnityYAMLMerge>
```

TortoiseGit

Go to Preferences > Diff Viewer > Merge Tool and click the *Advanced* button.
In the popup, type `.unity` in the extension field.
In the External Program field type:

```
<path to UnityYAMLMerge> merge -p %base %theirs %mine %merged
```

Then, follow the same procedure to add the `.prefab` extension.

PlasticSCM

Go to Preferences > Merge Tools and click the *Add* button.
Select *External* merge tool.
Select *Use with files that match the following pattern*.
Add the `.unity` extension.
Enter the command:

```
<path to UnityYAMLMerge> merge -p "@basefile" "@sourcefile"  "@destinationfile" "@output"
```

Then, follow the same procedure to add the `.prefab` extension.

SourceTree

Go to Tools > Options > Diff.
Select *Custom* in the Merge Tool dropdown.
Type the path to UnityYAMLMerge in the *Merge Command* text field.
Type `merge -p $BASE $REMOTE $LOCAL $MERGED` in the *Arguments* text field.

Troubleshooting The Editor

Leave feedback

The following sections explain how to troubleshoot and prevent problems with the Unity editor in different situations. In general, make sure your computer meets all the system requirements, it's up-to-date, and you have the required user permissions in your system. Also make backups regularly to protect your projects.

Versions

You can install different versions of the editor in different folders. However, make sure you backup your projects as these might be upgraded by a newer version, and you won't be able to open them in an older version of Unity. See the manual page on installing Unity for further information.

Licenses of add-ons are valid only for the Unity versions that share the same major number, for example 3.x and 4.x. If you upgrade to a minor version of Unity, for example 4.0 to 4.1, the add-ons will be kept.

Activation

Internet Activation is the preferred method to generate your license of Unity. But if you are having problems follow these steps:

Disconnect your computer from the network, otherwise you might get a "tx_id invalid" error.
Select Manual Activation.
Click on Save License Request.
Choose a known save location, for example the Downloads folder.
Reconnect to the network and open https://license.unity3d.com/
In the file field click Browse, and select the license request file.
Choose the required license for Unity and fill out the information requested.
Click Download License and save the file.
Go back to Unity and select Manual Activation if required.
Click on Read License and then select the downloaded license file.
If you still have problems with registering or logging in to your user account please contact support@unity3d.com.

Failure to Start

If Unity crashes when starting then firstly make sure that your computer meets the minimal system requirements. Also update to the latest graphic and sound drivers.

If you get disk write errors, you should check your user account restrictions. When in MacOS, note the "root user" is not recommended and Unity hasn't been tested in this mode. Unity should always have write permissions for its folders, but if you are granting them manually check these folders:

On Windows:

Unity's installation folder
`%AllUsersProfile%\Unity` (typically C:\ProgramData\Unity)
`C:\Documents and Settings\<user>\Local Settings\Application Data\Unity`
`C:\Users\<user>\AppData\Local\Unity`
MacOS:

Package contents of Unity.app

```
/Library/Application Support/Unity
~/Library/Logs/Unity
```

Some users have experienced difficulties when using hard disks formated with non-native partitions, and using certain software to translate data between storage devices.

Fonts

Corrupt fonts can crash Unity, you can find damaged files following these steps:

On Windows:

Open the fonts folder on your computer, located in the “Windows” folder.
Select “Details” from the “View” menu.
Check the “Size” column for fonts with a “0” size, which indicates a problematic file.
Delete corrupt fonts and reinstall them.

On MacOS:

Launch your Font Book application.
Select all the fonts.
Open the “File” menu and choose “Validate Fonts” -> problematic fonts will be shown as invalid.
Delete corrupt fonts and reinstall them.

The system might have resources constrained, for example running in a virtual machine. Use the Task Manager to find processes consuming lots of memory.

Corrupt Project or Installation

Unity could try to open a project that is corrupt, this might include the default sample project. In such case rename or move the folder of the project. After Unity starts correctly you can restore the project’s folder if wished.

In the event of a corrupt installation, you may need to reinstall Unity - see the instructions below.

In Windows, there could be problems like installation errors, registry corruption, conflicts, **etc**. For example, error 0xC0000005 means the program has attempted to access memory that it shouldn’t. If you added new hardware or drivers recently, remove and replace the hardware to determine if it’s causing the problem. Run diagnostics software and check information on trouble-shooting the operating system.

Performance and Crashes

If the editor runs slowly or crashes, particularly on builds, this might be caused by all of the available system resources being consumed. Close all other applications when you build the project. Clean up the system using its utilities, and consult the Task Manager (Windows) or Activity Monitor (MacOS) to find out if there are processes using lots of resources, for example memory. Sometimes virus protection software can slow down or even block the file system with its scanning process.

Project Loss

There are many factors that can destroy a project, you should **constantly backup your projects** to prevent unfortunate accidents. When in MacOS, activate the TimeMachine using an external hard disk reserved for this sole purpose. After a loss you can try any of the file recovery utilities that exist, but sometimes this is irreversible.

Re-installation

Follow these steps to reinstall the editor:

Uninstall Unity. When in MacOS, drag the Unity app to trash.

Delete these files if present:

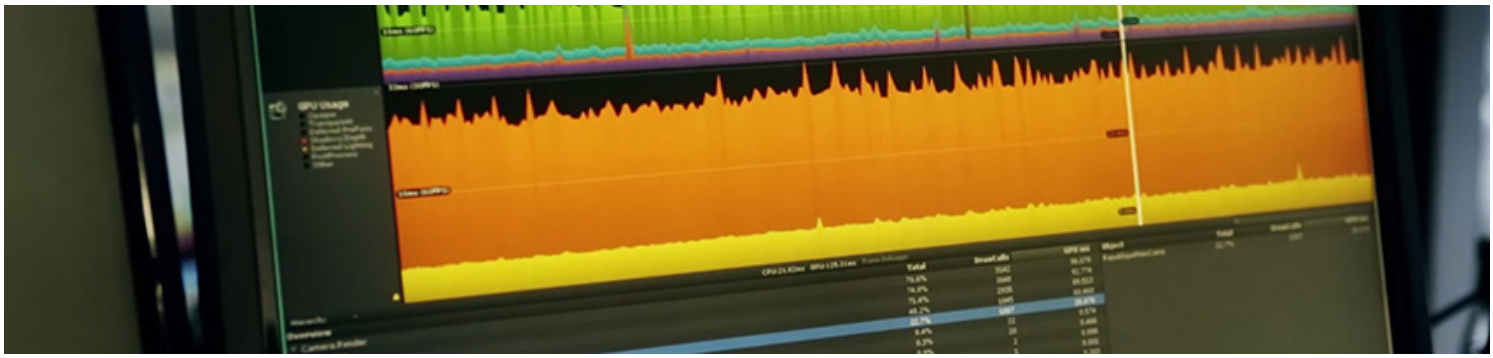
Windows:
`%AllUsersProfile%\Unity\` (typically C:\ProgramData\Unity)
MacOS:
`/Library/Application Support/Unity/`
Restart the computer.

Download the latest version from our website, since your original install might be corrupt:
http://unity3d.com/unity/download/archive

Reinstall Unity.

Advanced Development

Leave feedback



This section covers more advanced development techniques, which will be useful to developers and teams who are comfortable with the basics of developing in Unity.

The topics in this section give you more powerful control over working with **scenes**, managing **assets**, and streamlining your project.

Profiler overview

Leave feedback

The Unity **Profiler** Window helps you to optimize your game. It reports for you how much time is spent in the various areas of your game. For example, it can report the percentage of time spent **rendering**, animating or in your game logic.

You can analyze the performance of the GPU, CPU, memory, rendering, and audio.

To see the profiling data, you play your game in the Editor with Profiling on, and it records performance data. The Profiler window then displays the data in a **timeline**, so you can see the frames or areas that spike (take more time) than others. By clicking anywhere in the timeline, the bottom section of the Profiler window will display detailed information for the selected frame.

Note that profiling has to instrument your code (that is; add some instructions to facilitate the check). While this has a small impact on the performance of your game, the overhead is small enough to not affect the game framerate.

Tips on using the Tool

When using the profiling tool, focus on those parts of the game that consume the most time. Compare profiling results before and after code changes and determine the improvements you measure. Sometimes changes you make to improve performance might have a negative effect on frame rate; there may be unexpected consequences of your code optimization.

See Profiler window documentation for details of the Profiler window.

See also: Optimizing Graphics Performance.

Profiler window

Leave feedback

Access the Profiler window in the Unity Editor via the toolbar: **Window** > **Profiler**.

See Profiler overview for a summary of how the Profiler works.

Profile window

Profiler Controls

The Profiler controls are in the **toolbar** at the top of the window. Use these to turn profiling on and off, and navigate through profiled frames. The transport controls are at the far right end of the **toolbar**. Note that when the game is running and the profiler is collecting data, clicking on any of these transport controls pauses the game. The controls go to the first recorded frame, step one frame back, step one frame forward and go to the last frame respectively.

The Profiler does not keep all recorded frames, so the notion of the *first* frame should really be though of as the oldest frame that is still kept in memory. The "current" transport button causes the profile statistics window to display data collected in real-time. The Active Profiler popup menu allows you to select whether profiling should be done in the editor or a separate player (for example, a game running on an attached iOS device). Save button lets you write the recorded frames to a file. Correspondingly, Load button reads data saved earlier. You can also load a binary profile data written out by the player (when generating log, set Profiler.enableBinaryLog to enable binary format). If "Load" is clicked while the shift button is pressed, file contents is appended to the current profile frames in memory.

Deep Profiling

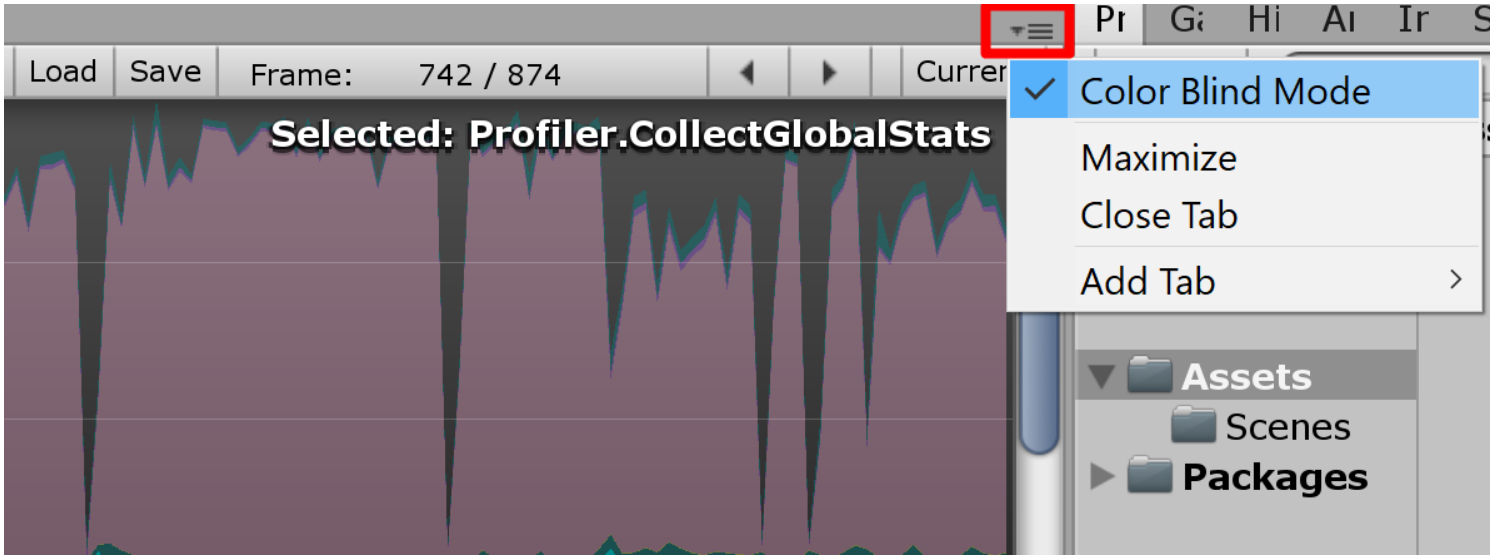
When you turn on **Deep Profile**, *all* your script code is profiled - that is, all function calls are recorded. This is useful to know where exactly time is spent in your game code.

Note that Deep Profiling incurs a **very large overhead** and uses a lot of memory, and as a result your game will run significantly slower while profiling. If you are using complex script code, Deep Profiling might not be possible at all. Deep profiling should work fast enough for small games with simple scripting. If you find that Deep Profiling for your entire game causes the frame rate

to drop so much that the game barely runs, you should consider not using this approach, and instead use the approach described below. You may find deep profiling more helpful as you are designing your game and deciding how to best implement key features. Note that for large games deep profiling may cause Unity to run out of memory and so for this reason deep profiling may not be possible.

Manually profiling blocks of your script code will have a smaller overhead than using Deep Profiling. Use Profiler.BeginSample and Profiler.EndSample scripting functions to enable and disable profiling around sections of code.

Color Blind Mode

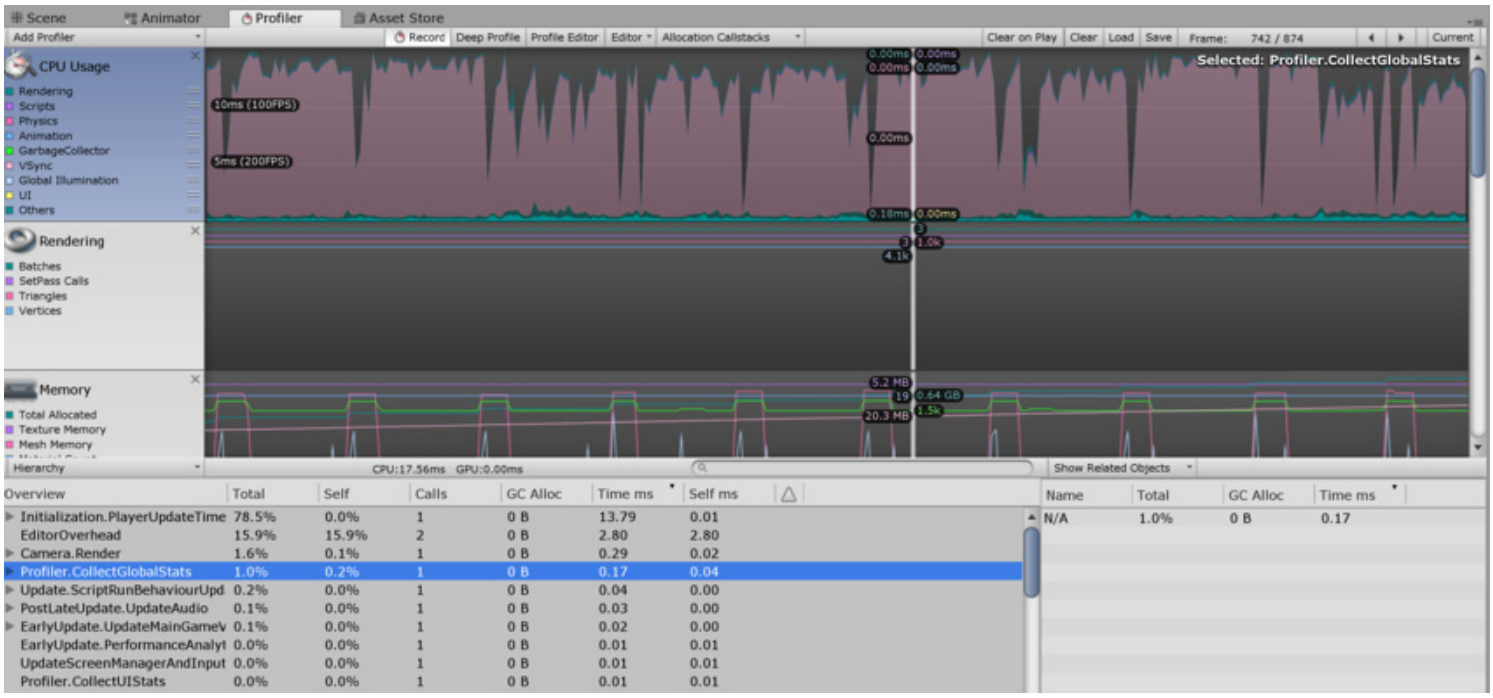


The Profiler window features a **Color Blind Mode**, which uses higher contrast colors in the graphs to enhance visibility for users with red-green color blindness (such as deuteranopia, protanopia, or tritanopia). To enable it, click the context menu in the upper-right corner of the Profiler window, and click **Color Blind Mode**.

View SyncTime

When running at a fixed framerate or running in sync with the vertical blank, Unity records the waiting time in "Wait For Target **FPS**". By default this amount of time is not shown in the profiler. To view how much time is spent waiting, you can toggle "View SyncTime". This is also a measure of how much headroom you have before losing frames.

Profiler Timeline



The upper part of the Profiler window displays performance data over time. When you run a game, data is recorded each frame, and the history of the last several hundred frames is displayed. Clicking on a particular frame will display its details in the lower part of the window. Different details are displayed depending on which **timeline** area is currently selected.

The vertical scale of the timeline is managed automatically and will attempt to fill the vertical space of the window. Note that to get more detail in say the CPU Usage area you can remove the Memory and **Rendering** areas. Also, the splitter between the timeline and the statistics area can be selected and dragged downward to increase the screen area used for the timeline chart.

The timeline consists of several areas: CPU Usage, Rendering and Memory. These areas can be removed by clicking the close button in the panel, and re-added again using the *Add Area* drop down in the Profile Controls bar.

Note that the coloured squares in the label area can control whether the associated timeline is displayed or not. To remove a sample from the display click on the colour key. The key will dim and the data will be removed from the graph. This can be useful to identify the cause of spikes in the CPU graph, for example.

WebGL

You can use the Unity profiler on **WebGL**, just like on any other platform. One important distinction is that you cannot attach to running players in **WebGL**, though, as **WebGL** uses WebSockets for communication, which will not allow incoming connections on the browser side. Instead, you need to use the “Autoconnect profiler” checkbox in the build settings. Note also that draw calls cannot currently be profiled for **WebGL**.

Remote Profiling

To profile your game running on another device or a Unity player running on another computer, you can connect the Unity Editor to that other device or computer. The dropdown **Active Profiler** shows all Unity players running on the local network. These players are identified by player type and the host name running the player “iPhonePlayer (Toms iPhone)”.

To be able to connect to a Unity player, you must launch that Unity player as a **Development build** (menu: **File** > **Build Settings...**).

Check the **Development Build** option in the dialog box. From here you can also check **Autoconnect Profiler** to make the Editor and Player Autoconnect at startup.

iOS

Enable remote profiling on **iOS** devices by following these steps:

Connect your iOS device to your WiFi network. (The Profiler uses a local WiFi network to send profiling data from your device to the Unity Editor.)
In the Unity Editor's **Build Settings** dialog box (menu: **File** > **Build Settings...**), check the **Autoconnect Profiler** checkbox.
Attach your device to your Mac via cable. In the Unity Editor's **Build Settings** dialog box (menu: **File** > **Build Settings...**), check the **Autoconnect Profiler** checkboxcheck and select **Build & Run**.
When the app launches on the device, open the Profiler window in the Unity Editor (**Window** > **Analysis** > **Profiler**).
If you are using a firewall, you need to make sure that ports 54998 to 55511 are open in the firewall's outbound rules - these are the ports used by Unity for remote profiling.

Note: Sometimes the Unity Editor might not autoconnect to the device. In such cases you can initiate the Profiler connection from Profiler window **Active Profiler** drop down menu by select appropriate device.

Android

There are two methods to enable remote profiling on Android devices: WiFi or ADB.

For WiFi profiling, follow these steps:

Make sure to disable Mobile Data on your Android device.
Connect your Android device to your WiFi network.(The Profiler uses a local WiFi network to send profiling data from your device to the Unity Editor.)
Attach your device to your Mac or PC via cable. Check the **Development Build** and **Autoconnect Profiler** checkboxes in Unity's Build Settings dialog box, and click on **Build & Run** in the Unity Editor.
When the app launches on the device, open the Profiler window in the Unity Editor (Menu: **Window** > **Analysis** > **Profiler**).
If the Unity Editor fails to autoconnect to the device, select the appropriate device from the Profiler window **Active Profiler** drop down menu.
Note: The Android device and host computer (running the Unity Editor) must both be on the same subnet for the device detection to work.

For ADB profiling, follow these steps:

Attach your device to your Mac or PC via cable and make sure ADB recognizes the device (i.e. it shows in *adb devices* list).
In the Unity Editor's **Build Settings** dialog box (menu: **File** > **Build Settings**), check the **Development Build** checkboxcheck and select **Build & Run**.
When the app launches on the device, open the Profiler window in the Unity Editor (Menu: **Window** > **Analysis** > **Profiler**).
Select the *AndroidProfiler(ADB@127.0.0.1:34999)* from the Profiler Window **Active Profiler** drop down menu.
Note: The Unity Editor automatically creates an adb tunnel for your application when you click on **Build & Run**. If you want to profile another application or you restart the adb server you have to setup this tunnel manually. To do this, open a Terminal window / CMD prompt and enter:

```
adb forward tcp:34999 localabstract:Unity-{insert bundle identifier here}
```

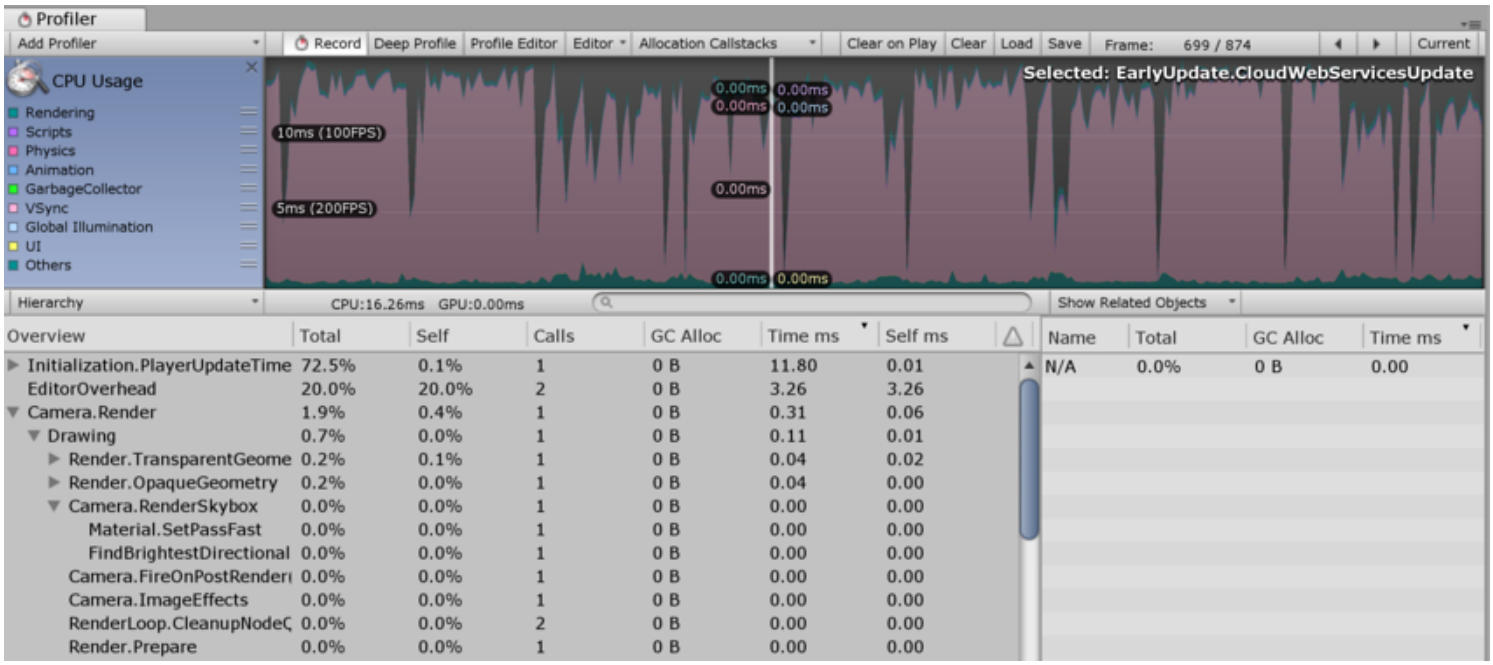
Note: The entry in the drop down menu is only visible when the selected target is Android.

If you are using a firewall, you need to make sure that ports 54998 to 55511 are open in the firewall's outbound rules - these are the ports used by Unity for remote profiling.

- 2017–05–16 Page amended with no editorial review

CPU Usage Profiler

Leave feedback

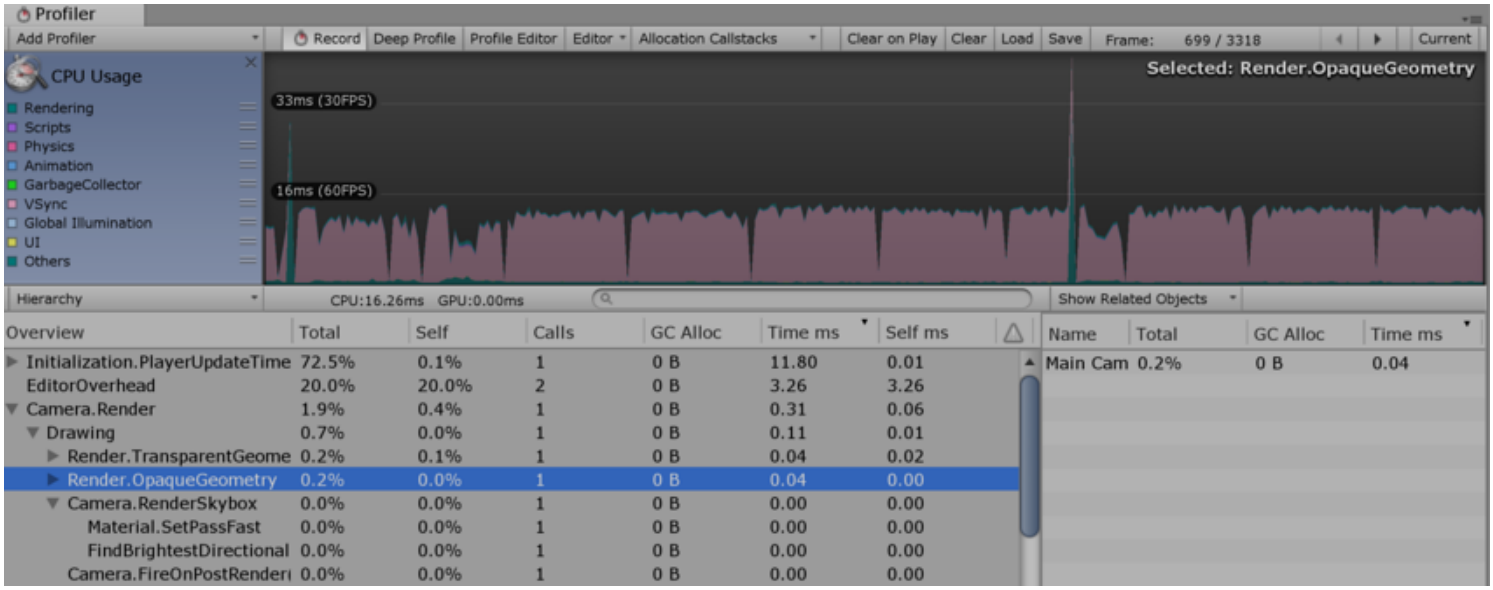


The CPU Usage Profiler displays where time is spent in your game. When it is selected, the lower pane displays hierarchical time data for the selected frame. See documentation on the Profiler Window to learn more about the information on the Profiler timeline.

Hierarchy mode: Displays hierarchical time data.
Group Hierarchy mode: Groups time data into logical groups (such as Rendering, Physics, Scripts). Because children of any group can also be in different groups (for example, some scripts might also call rendering functions), the percentages of group times often add up to more than 100%.
Drag chart labels up and down to reorder the way the CPU chart is stacked.

Selecting individual items

When an item is selected in the lower pane, its contribution to the CPU chart is highlighted (and the rest are dimmed). Clicking on an item again de-selects it.



Render.OpaqueGeometry is selected and its contribution is highlighted in the chart

In the hierarchical time data, the **Self** column refers to the amount of time spent in a particular function, not including the time spent calling sub-functions. In the screenshot above, 41.1% of time is spent in the `Camera.Render` function. This function does a

lot of work, and calls the various drawing and culling functions. Excluding all of these functions, only 2.1% of time is spent in the `Camera.Render` function itself.

The **Time ms** and **Self ms** columns show the same information but in milliseconds. `Camera.Render` takes 0.01ms but, including all the functions it calls, 0.21ms are consumed. The **GC Alloc** column shows how much memory has been allocated in the current frame, which is later collected by the garbage collector. Keep this value at zero to prevent the garbage collector from causing hiccups in your framerate.

The **Others** section of the CPU profiler records the total of all areas that do not fall into **Rendering**, **Scripts**, Physics, Garbage Collection or VSync. This includes Animation, AI, Audio, Particles, **Networking**, Loading, and PlayerLoop.

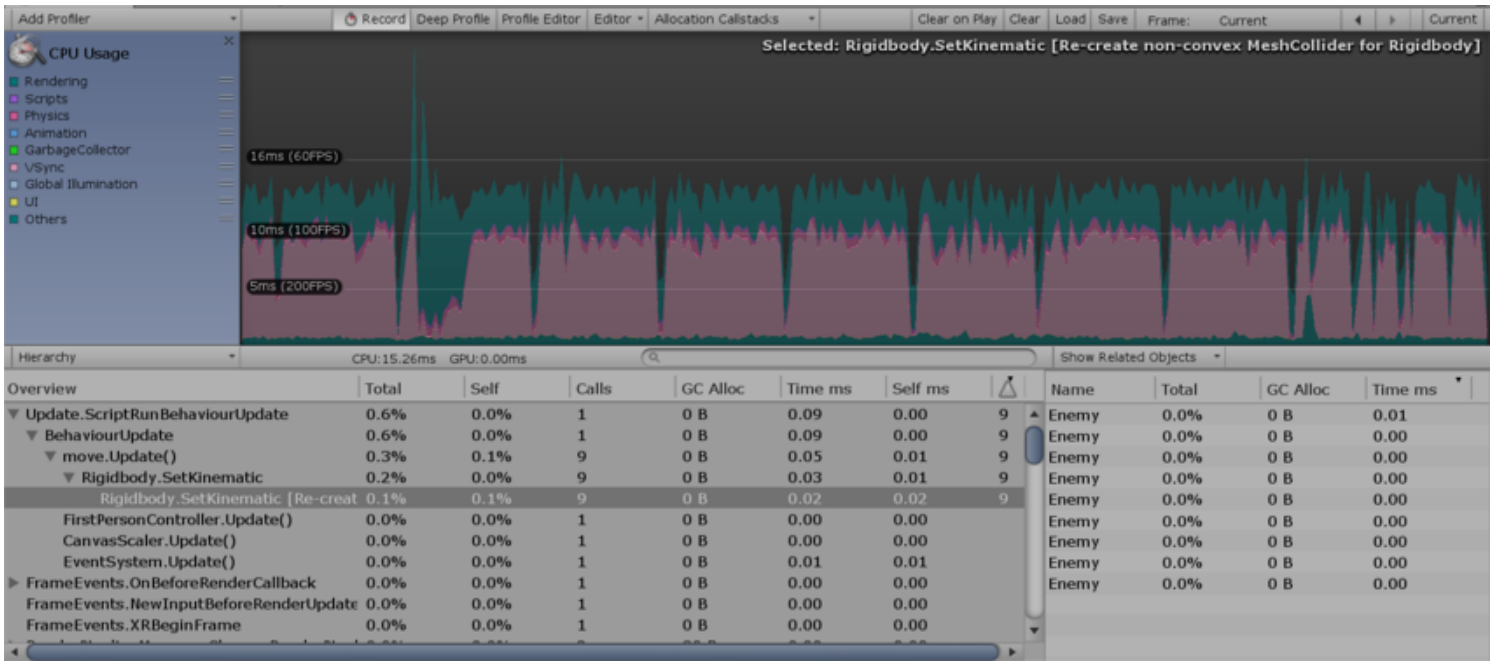
Physics markers

The descriptions below provide a brief account of what each of the various high-level Physics Profiler markers mean.

Physics.Simulate: Called from `FixedUpdate`. This updates the present state of the physics by instructing the **physics engine** (PhysX) to run its simulation.
Physics.Processing: Called from `FixedUpdate`. This is where all the non-cloth physics jobs are processed. Expanding this marker shows low-level detail of the work being done internally in the physics engine.
Physics.ProcessingCloth: Called from `FixedUpdate`. This is where all the cloth physics jobs are processed. Expanding this marker will show low level detail of the work being done internally in the physics engine.
Physics.FetchResults: Called from `FixedUpdate`. This is where the results of the physics simulation are collected from the physics engine.
Physics.UpdateBodies: Called from `FixedUpdate`. This is where all the physics bodies have their positions and rotations updated as well as where messages that communicate these updates are sent.
Physics.ProcessReports: Called from `FixedUpdate`. This stage is run once the physics `FixedUpdate` has concluded, and is where all the various stages of responding to the results of the simulation are processed. Contacts, **joint** breaks and triggers are updated and messaged here. There are four distinct sub stages:
Physics.TriggerEnterExits: Called from `FixedUpdate`. This is where `OnTriggerEnter` and `OnTriggerExit` events are processed.
Physics.TriggerStays: Called from `FixedUpdate`. This is where `OnTriggerStay` events are processed.
Physics.Contacts: Called from `FixedUpdate`. This is where `OnCollisionEnter`, `OnCollisionExit` and `OnCollisionStay` events are processed.
Physics.JointBreaks: Called from `FixedUpdate`. This is where updates and messages relating to joints being broken is processed.
Physics.UpdateCloth: Called from `Update`. This is where updates relating to cloth and their skinned meshes are made.
Physics.Interpolation: Called from `Update`. This stage deals with the interpolation of positions and rotations for all the physics objects.

Performance warnings

There are some common performance issues the CPU Profiler is able to detect and warn you about. These appear in the **Warning** column of the lower pane when viewing the **CPU Usage**.



A Profiler warning indicating that Static Colliders have been moved

The specific issues the Profiler can detect are:

Rigidbody.SetKinematic [Re-create non-convex MeshCollider for Rigidbody]
Animation.DestroyAnimationClip [Triggers RebuildInternalState]
Animation.AddClip [Triggers RebuildInternalState]
Animation.RemoveClip [Triggers RebuildInternalState]
Animation.Clone [Triggers RebuildInternalState]
Animation.Deactivate [Triggers RebuildInternalState]

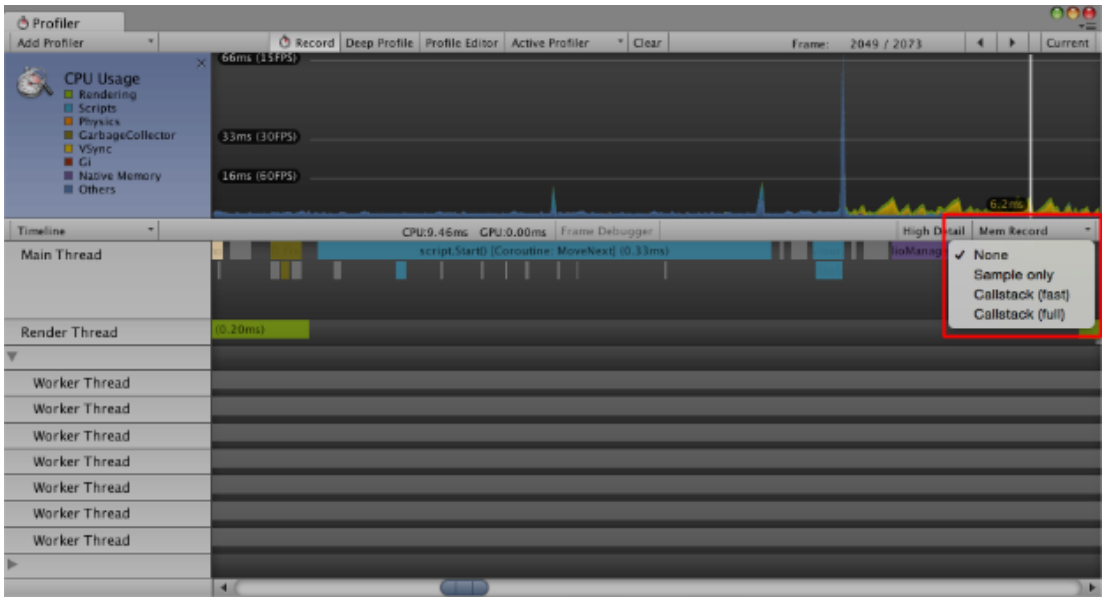
In the screenshot above, the Profiler is showing the **Static** Collider**.Move** warning. The **Warning** column shows that this warning has been triggered 12 times in the current frame. The term “delayed cost” means that, although the entry in the Profiler may show a low cost (in this case 0.00ms), the action may trigger more system-demanding operations later on.

CPU Profiler Timeline

Mem Record: Native memory performance profiling

Native memory performance profiling allows you to profile activity inside Unity’s native memory management system and assess how it is affecting runtime performance. This can be useful when searching for unwanted or resource-intensive allocation patterns in Unity’s memory management.

To profile Unity’s native memory management, you need to record it. To access native memory recording mode (called **Mem Record** in Unity), go to **Window** > **Analysis** > **Profiler** to open the Profiler window. Select the **CPU Usage** Profiler (if it is not visible, click **Add Profiler** > **CPU**) then the drop-down menu underneath the Profiler. Next, click **Timeline** and then select **Mem Record**.

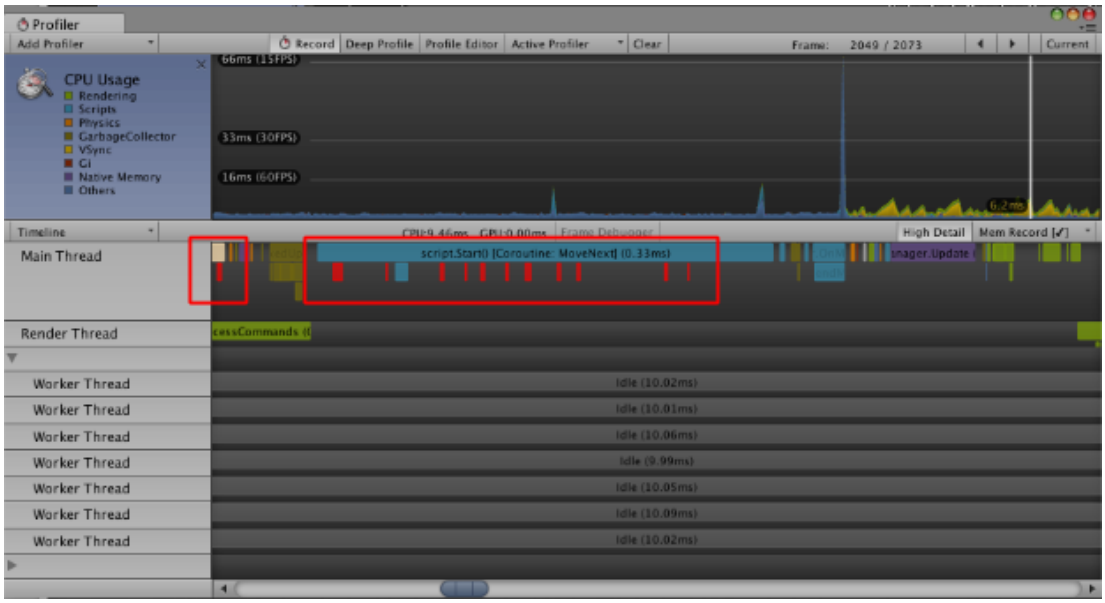


Selecting recording mode

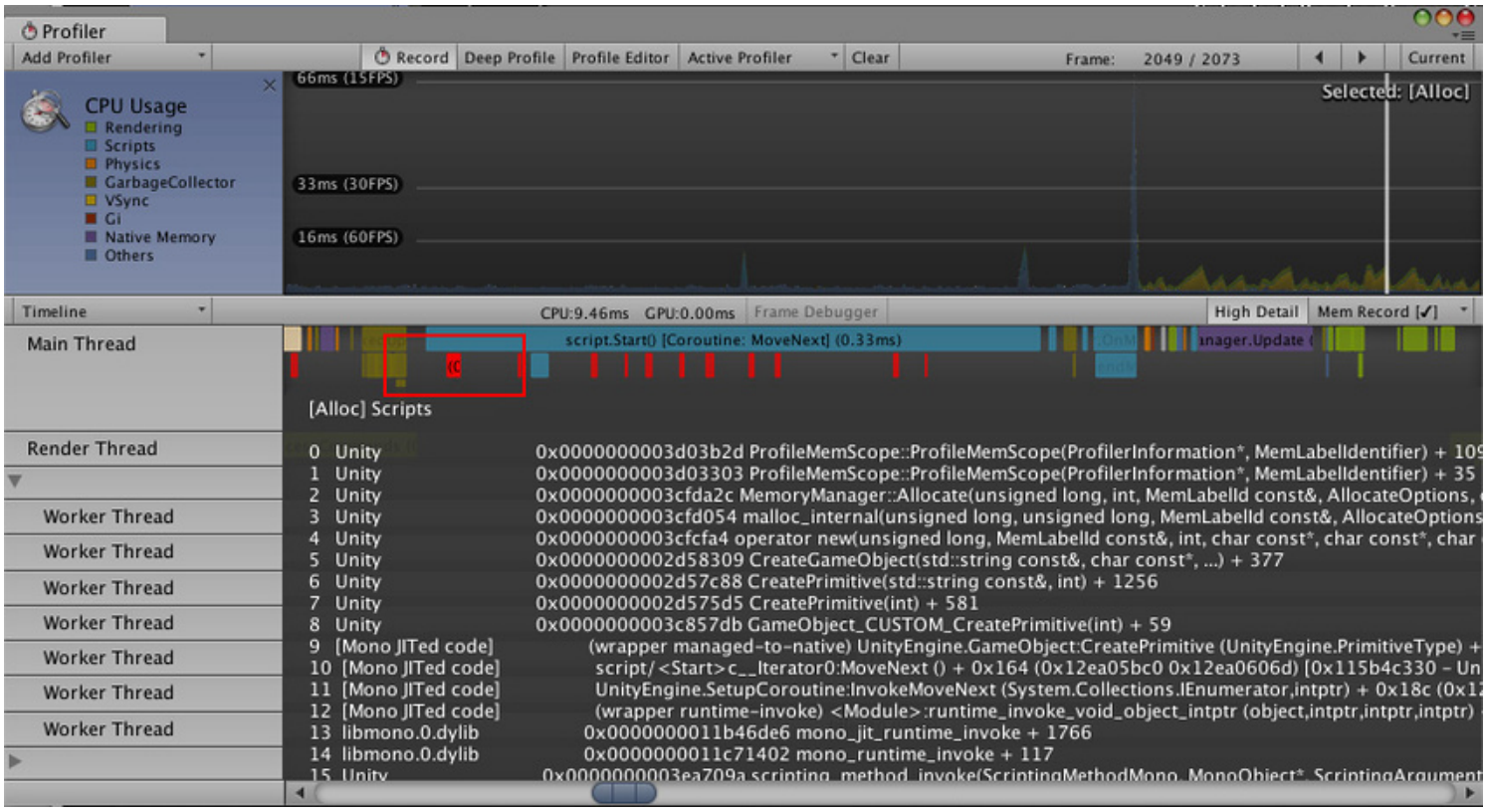
Option	Function	Impact on performance
None	Mode disabled. This is the default selection.	N/A
Sample only	Records memory allocations, re-allocations, de-allocations, activity type, and system.	Low
Callstack (fast)	This has the same functionality as **Sample only**, but also records a shortened callstack from the native allocation site to where the callstack transitions from native symbols into script symbols. Effectively, you can only see the callstack up to the deepest script symbol.	Medium
Callstack (full)	This has the same functionality as **Sample only**, but also records the callstack with full script-to-native and native-to-script transitions.	High

Note: When the active Profiler is only connected to a standalone player, only the low-impact **Sample only** mode is supported.

The recorded memory allocation samples appear in the Profiler window in bright red.



Click the **High Detail** button next to **Mem Record** to enable High Detail mode. Select a sample to display the allocation type and system. If the callstack was recorded for the selected allocation sample, the associated callstack symbols are resolved and displayed as well:



Using Mem Record

There are a number of instances where the **Mem Record** function is useful. For example:

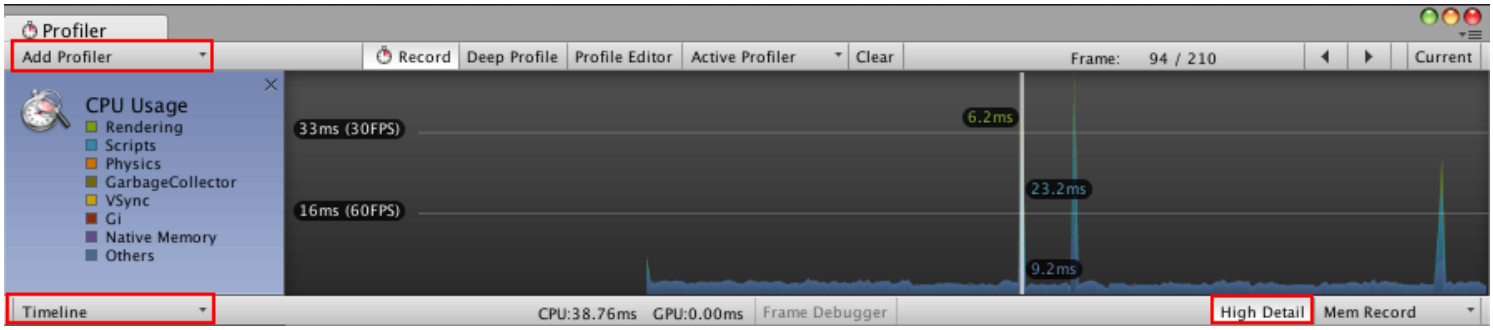
Learning when a system is doing many small allocations instead of just a few large ones.
Learning when a Worker Thread accidentally allocates memory (for example by unintended MemLabel use).
Finding lock contention (when several threads try to access the native memory system simultaneously).
Finding sources of memory fragmentation (particularly important for low-memory devices).

High Detail view of Timeline

The **High Detail** view for the CPU Usage Profiler Timeline gives at least one **pixel** of width to every time sample recorded by Unity's CPU Usage Profiler.

This allows you to see a complete overview of all activity in a frame, including short-lived activities such as thread synchronization or memory allocation.

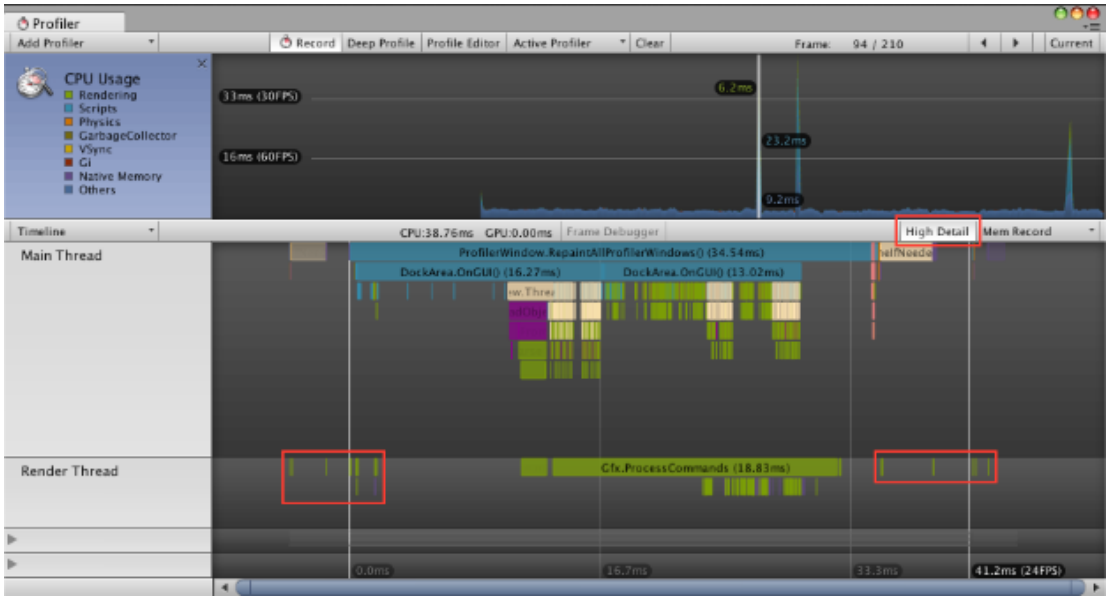
To enable the **High Detail** view, go to **Window** > **Analysis** > **Profiler** to open the Profiler window. Select the **CPU Usage** Profiler (if it is not visible, click **Add Profiler** > **CPU**) then select the drop-down menu underneath the Profiler and click **Timeline** followed by **High Detail**.



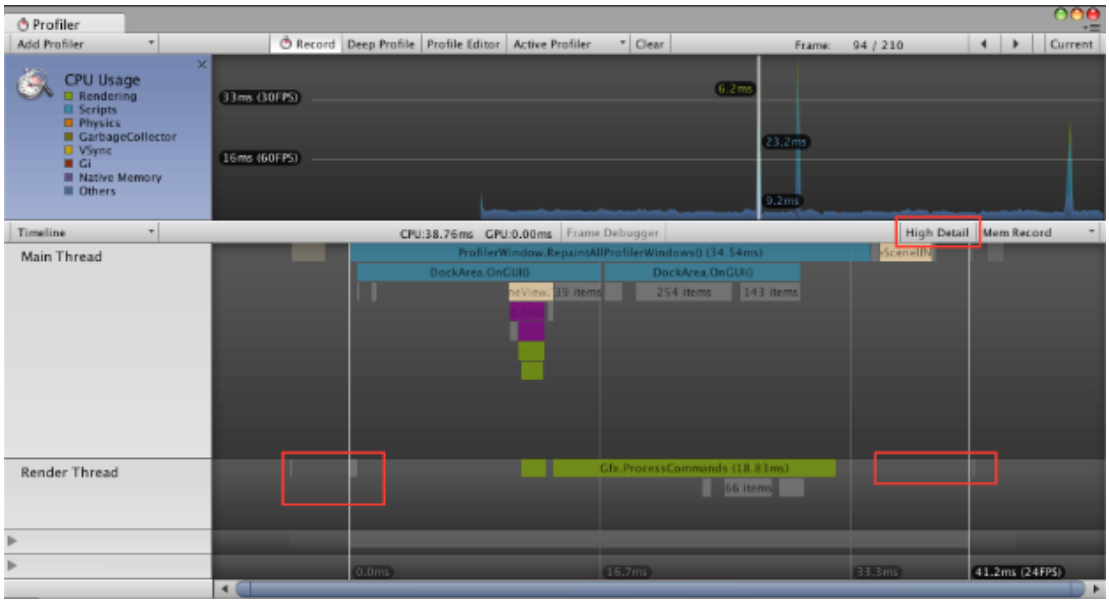
Comparison

The following two images show the difference between the **High Detail** view and the normal view for the CPU Usage Profiler's Timeline.

High Detail view

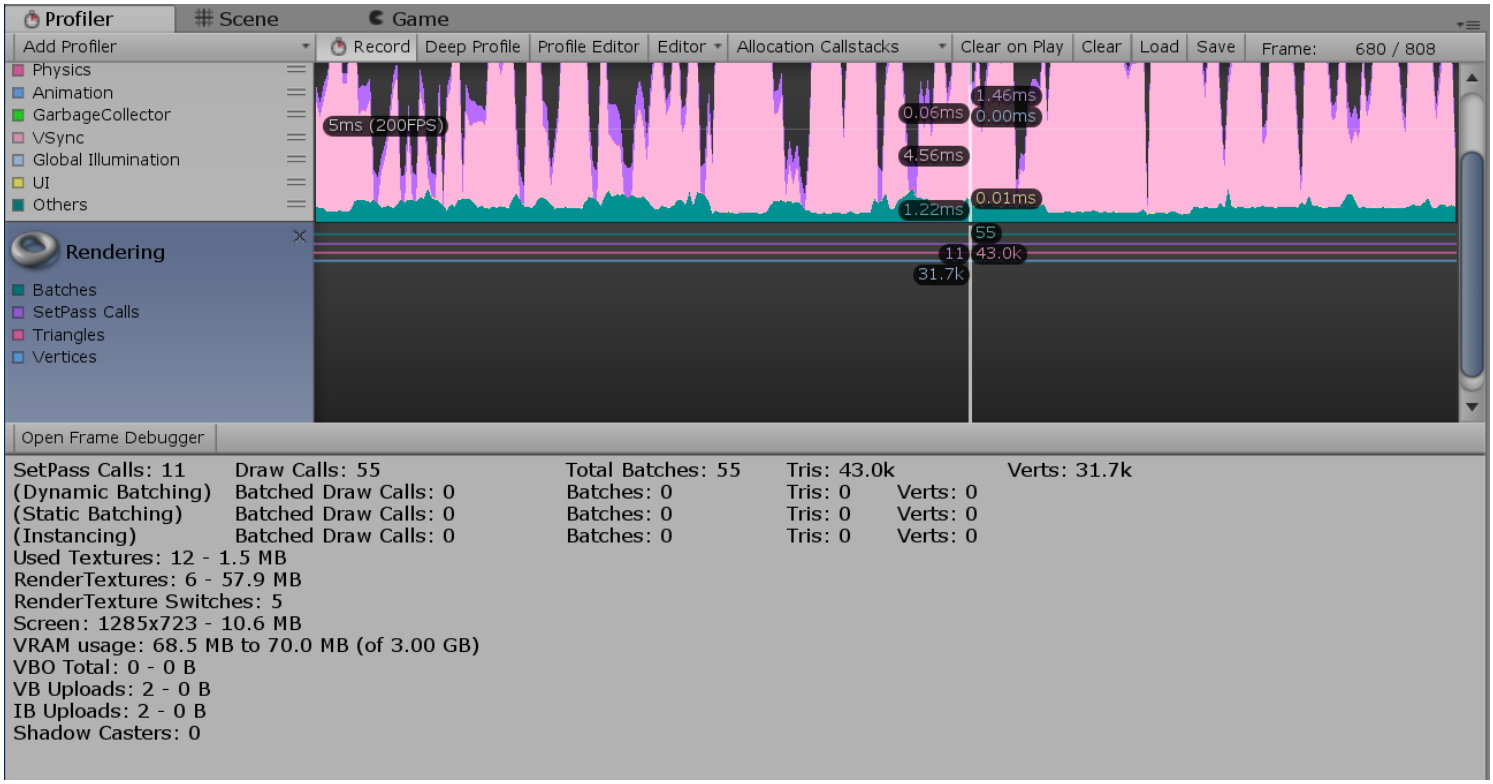


Normal view



Rendering Profiler

Leave feedback



The **Rendering Profiler** displays **rendering** statistics. The **timeline** displays the number of Batches, SetPass Calls, Triangles and Vertices rendered. The lower pane displays more **rendering** statistics, which closely match the ones shown in the GameView Rendering Statistics window.

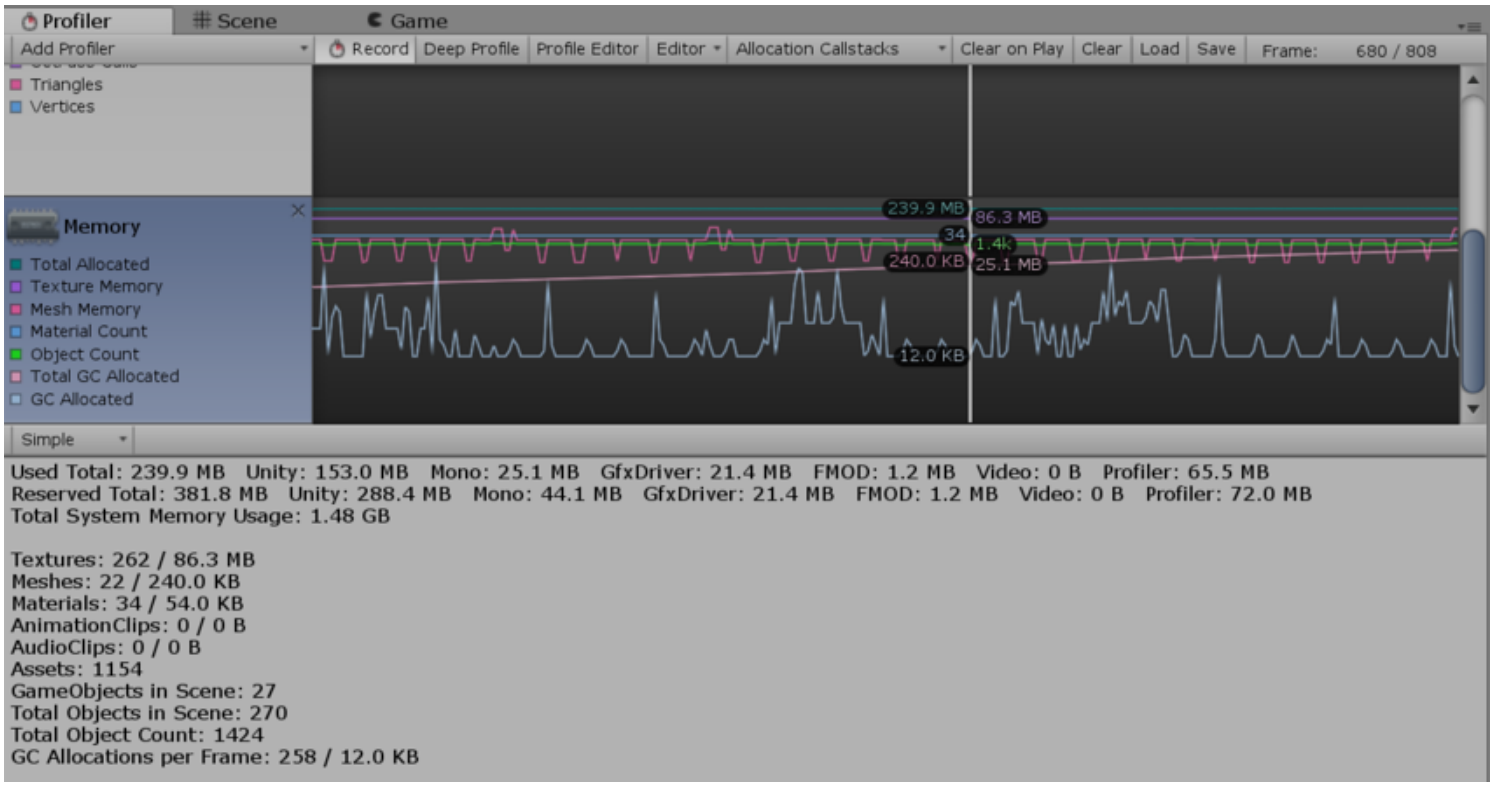
Memory Profiler

Leave feedback

There are two modes you can use in the Memory **Profiler** to inspect the memory usage of your application. This is selected in the dropdown in the top-left of the lower panel.

Simple

The **Simple** view shows a simple overview how memory is used throughout Unity in real-time on a per-frame basis.



Unity reserves memory pools for allocations in order to avoid asking the operating system for memory too often. This is displayed as a reserved amount, and how much is used.

The areas covered by this are:

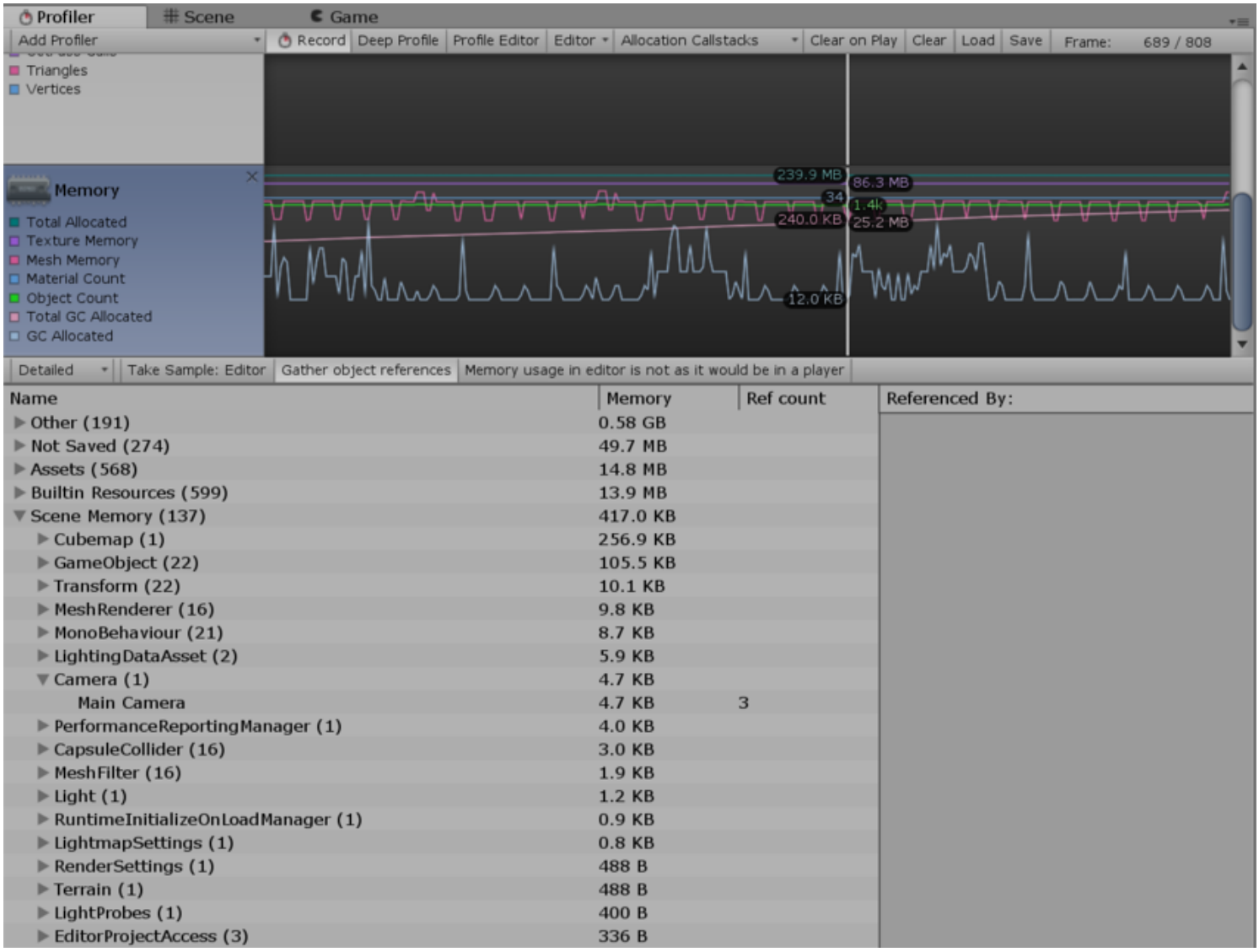
Unity The amount of memory tracked by allocations in native Unity code
Mono The total heap size and used heap size used by managed code. This memory is garbage-collected
GfxDriver The estimated amount of memory the driver is using on Textures, render targets, **Shaders** and **Mesh** data.
FMOD The Audio driver's estimated memory usage
Profiler Memory used for the Profiler data
The numbers that are displayed are not the same as the Task Manager or Activity Monitor, because some usage is untracked by the Memory Profiler. This includes memory used by some drivers, and memory used for executable code.

Memory statistics are shown for some of the most common **Asset**/object types. These stats include the count and the used memory (main and video memory):

Textures
Meshes
Materials
Animations
Audio
Object Count
Object Count is the total number of objects that are created. If this number rises over time, your game is creating some objects that are never destroyed.

Detailed

The **Detailed** view allows you take a snapshot of the current state. Use the **Take Sample** button to capture detailed memory usage. Obtaining this data takes some time, so the Detailed view should not be expected to give you real-time details. After taking a sample, the Profiler window is updated with a tree view where you can explore memory usage.



This displays individual Assets and **GameObject** memory usage. It also displays a reason for a **GameObject** to be in memory. Common reasons include:

Assets: Asset referenced from user or native code

Built-in Resources: Unity Editor resources or Unity default resources
Not Saved: GameObjects marked as DontSave
Scene Memory: GameObject and attached components
Other: GameObjects not marked in the above categories
Click on a GameObject in the list to view it in either the Project or the **Scene view**.

When profiling in the Editor, all numbers displayed by the Memory Profiler indicate the memory usage in the Editor. These are generally larger than when running in a player, because running the Unity Editor adds extra memory. For more precise numbers and memory usage for your app, use the Profiler connection to connect to the running player. This will give the actual usage on the target device.

Memory reported under `System.ExecutableAndDlls` is read-only memory, so the operating system might discard these pages as needed and later reload them from the file system. This generates lower memory usage, and usually does not directly contribute to the operating system's decision to kill the application. Some of these pages might also be shared with other applications that are using the same frameworks.

Audio Profiler

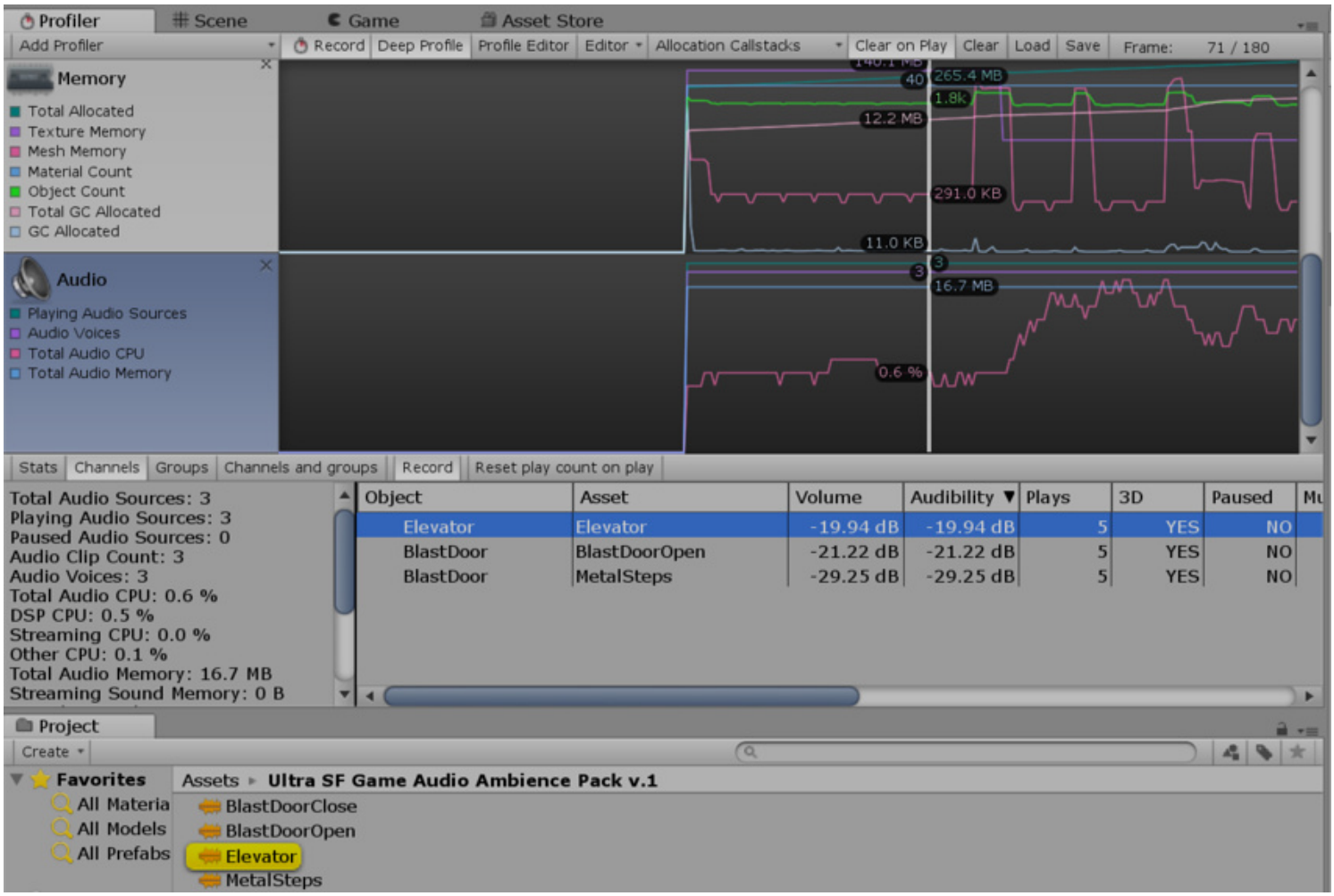
Leave feedback

In the **Profiler** window there is a pane called Audio. The pane monitors significant performance meters about the audio system, such as total load and voice counts. When you highlight the pane, the lower part of the window changes into a detailed view about various parts of the audio system not covered by the graphs.

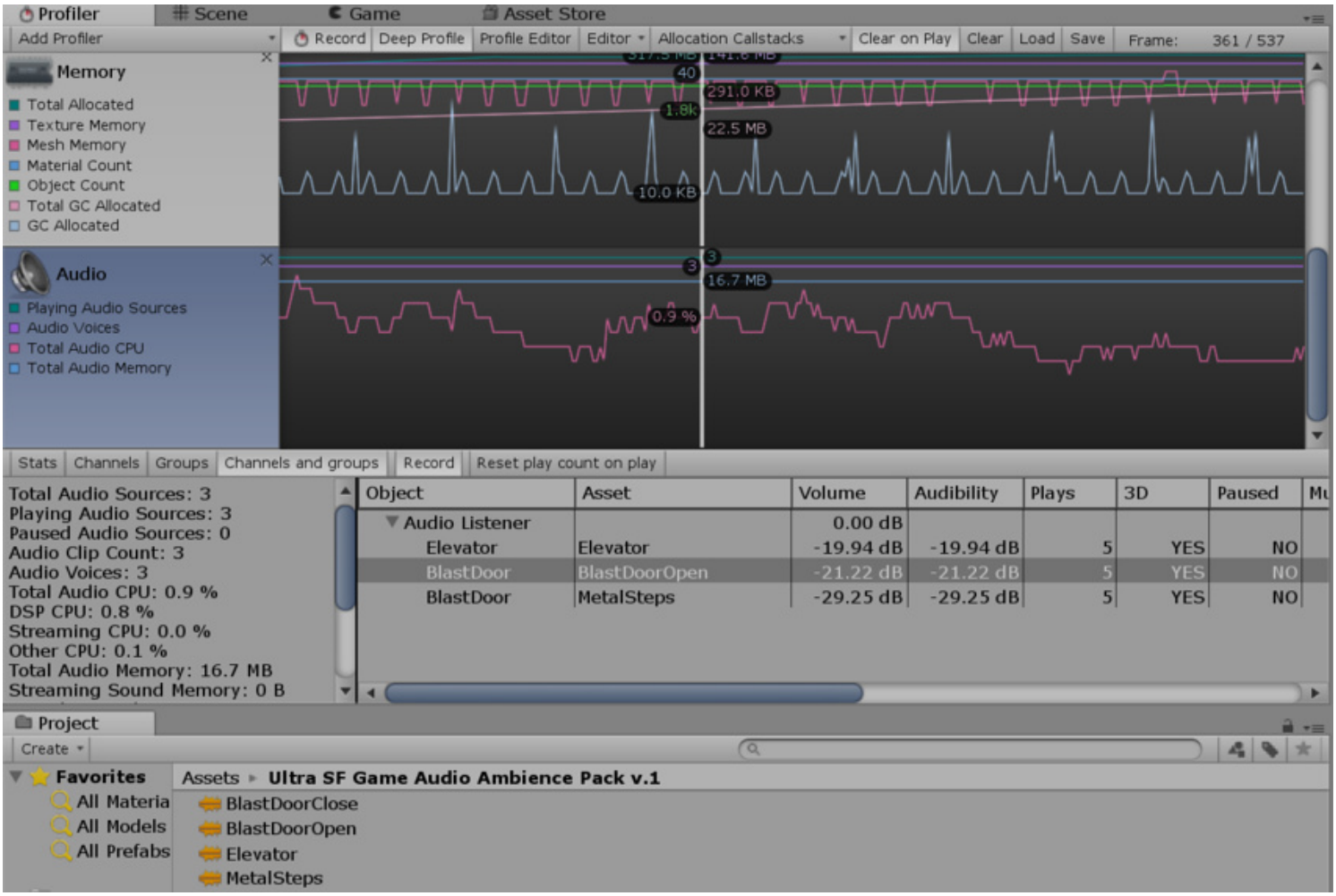
Playing Sources is the total playing sources in the **scene** at a specific frame. Monitor this to see if audio is overloaded.
Paused Sources is the total paused sources in the scene at a specific frame.
Audio Voice is the actually number of audio (FMOD channels) voices used. PlayOneShot is using voices not shown in Playing Sources.
Audio Memory is the total RAM used by the audio engine.
CPU usage can be seen in the bottom. Monitor this to see if Audio alone is taking up too much CPU.

Click the **Channels**, **Groups** or **Channels and groups** buttons for detailed per-frame logging of sound events. Here these events can be obtained and scrubbed, just like the renderer and memory graphs.

The rows in the frame log reveal information such as which **audio sources** played which clips, the volume at which they were played, the distance to the listener, and relative playback time. Clicking on one of these rows highlights the associated audio source and clip in the Project browser and Hierarchy window.



Channel view. When clicking a row the AudioClip **Asset** is highlighted first, then the AudioSource in the Hierarchy that played it.



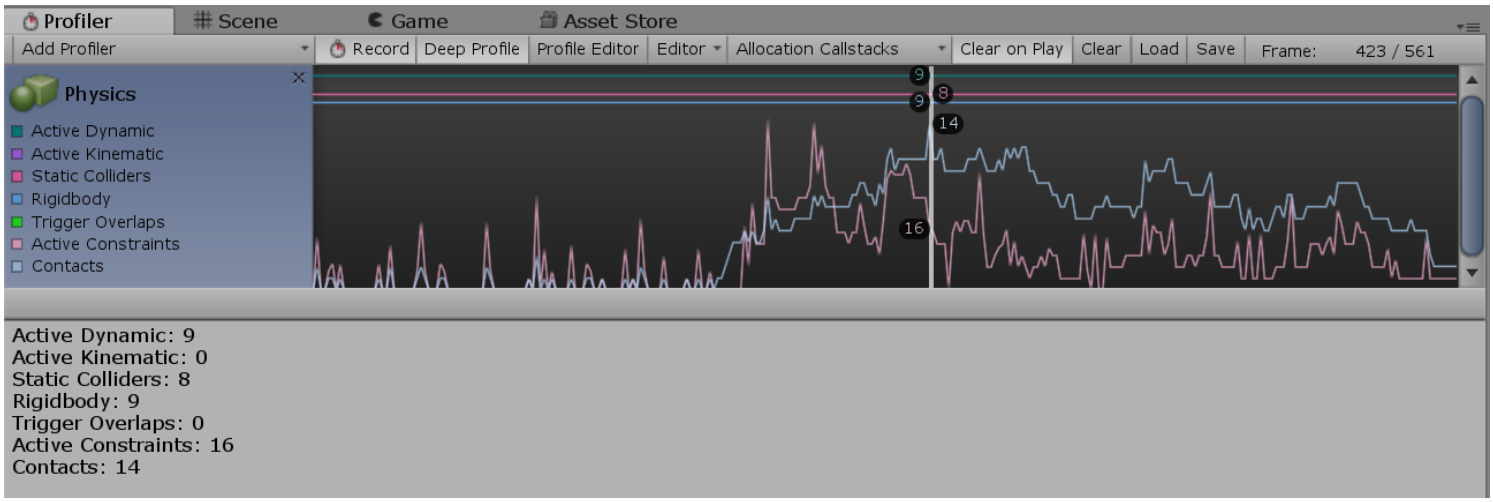
Channels and groups view. Here the AudioSource that played the sound in the selected row is highlighted.

Physics Profiler

Leave feedback

The Physics **Profiler** displays statistics about physics that have been processed by the **physics engine** in your **Scene**. This information can help you diagnose and resolve performance issues or unexpected discrepancies related to the physics in your **Scene**.

See also Physics Debug Visualization.



Physics Profiler

Properties

Property	Function
Active Dynamic	The number of non-Kinematic **Rigidbody components** that are not sleeping.
Active Kinematic	The number of Kinematic Rigidbody components that are not sleeping. Note that Kinematic Rigidbody components with **joints** attached may be processed multiple times per frame, and this contributes to the number shown. A Kinematic Rigidbody is active when MovePosition or MoveRotation is called in a frame, and remains active in the next frame.
Static Colliders	The number of **Collider** components on **GameObjects** that don't have Rigidbody components attached to the **GameObjects** or their parent **GameObjects**. If such **GameObjects** or their parent **GameObjects** have Rigidbody components, the Colliders will not count as Static Colliders. Those Colliders are instead referred to as Compound Colliders. These help to arrange multiple Colliders of a body in a convenient way, rather than having all the Colliders on the same GameObject as the Rigidbody component.
Rigidbody	The number of Rigidbody components processed by the physics engine, irrespective of the components' sleeping state.
Trigger Overlaps	The number of overlapping triggers (counted in pairs).
Active Constraints	The number of primitive constraints processed by the physics engine. Constraints are used as a building block of Joints as well as **collision** response. For example, restricting a linear or rotational degree of freedom of a ConfigurableJoint involves a primitive constraint per each restriction.

Property	Function
Contacts	The total number of contact pairs between all Colliders in the Scene, including the amount of trigger overlap pairs as well. Note that contact pairs are created per Collider pair once the distance between them is below a certain user configurable limit, so you may see contacts generated for Rigidbody components that are not yet touching or overlapping. Refer to Collider.contactOffset and ContactPoint.separation for more details.

Notes:

The numbers might not correspond to the exact number of GameObjects with physics components in your Scene. This is because some physics components are processed at a different rate depending on which other components affect it (for example, an attached Joint component). To calculate the exact number of GameObjects with specific physics components attached, write a custom script using the FindObjectsOfType function.

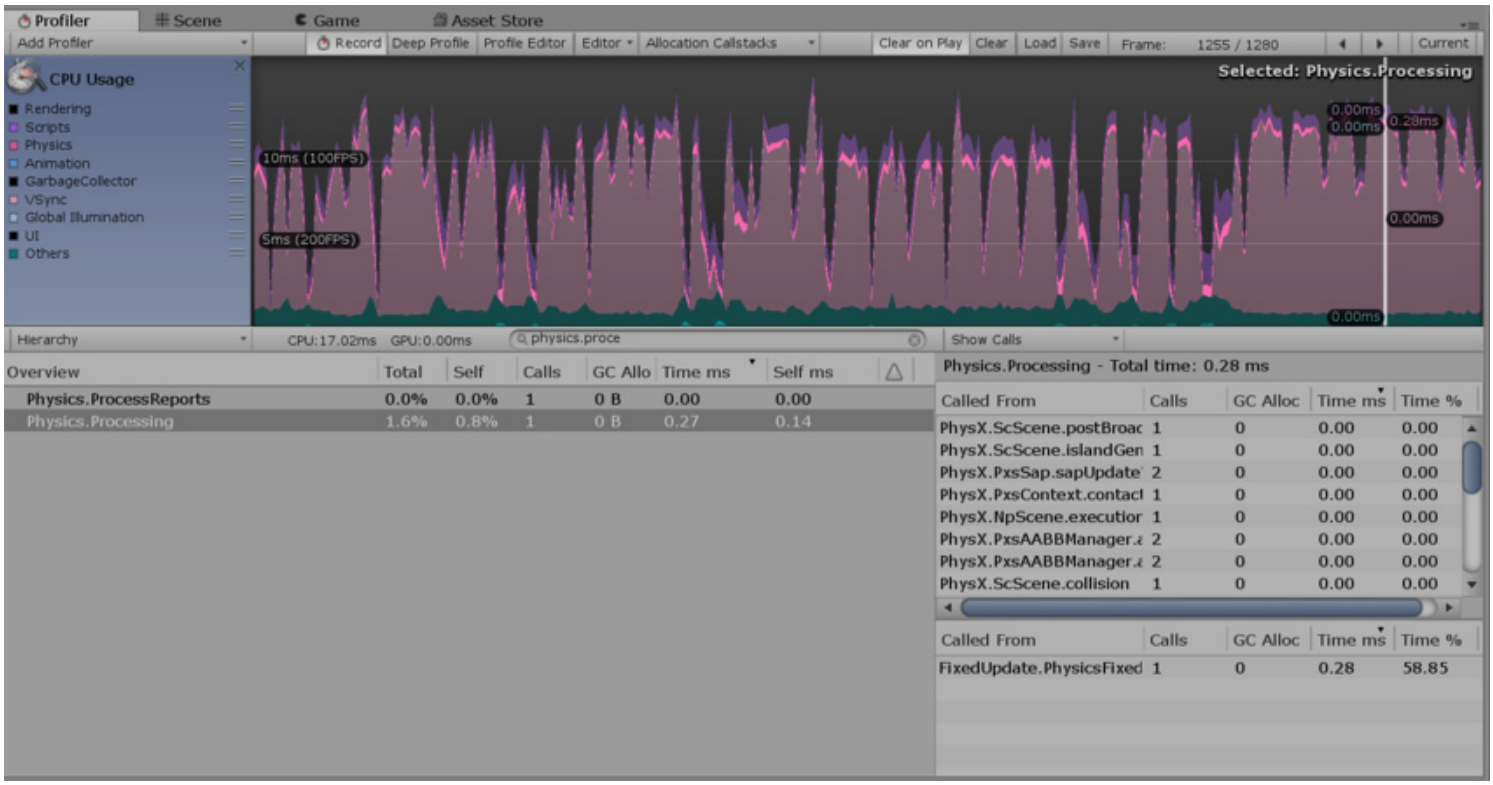
The Physics Profiler does not show the number of sleeping Rigidbody components. These are components which are not engaging with the physics engine, and are therefore not processed by the Physics Profiler. Refer to Rigidbody overview: Sleeping for more information on sleeping Rigidbody components.

Using the Physics Profiler to understand performance issues

The physics simulation runs on a separate fixed frequency update cycle from the main logic's update loop, and can only advance time via a `Time.fixedDeltaTime` per call. This is similar to the difference between `Update()` and `FixedUpdate()` (see documentation on the Time Manager for more).

When a heavy logics or graphics frame occurs that takes a long amount of time, the Physics Profiler has to call the physics simulation multiple times per frame. This means that an already resource-intensive frame takes even more time and resources, which can easily cause the physics simulation to temporarily stop due to the **Maximum Allowed Timestep** value (which can be set in **Edit** > **Project Settings** > **Time**).

You can detect this in your project by selecting the CPU Usage Profiler and checking the number of calls for **Physics.Processing** or **Physics.Simulate** in the **Overview** section.



Physics Profiler with the value of 1 in the **Calls** column

In this example image, the value of 1 in the **Calls** column indicates that the physics simulation was called one time over the last logical frame.

A call count close to 10 might indicate an issue. As a first solution, reduce the frequency of the physics simulation; if the issue continues, check what could have caused the heavy frame right before the Physics Profiler had to use that many simulation calls in order to catch up with the game time. Sometimes, a heavy graphics frame may cause more physics simulation calls later on in a Scene.

For more detailed information about the physics simulation in your Scene, click the triangle arrow to expand **Physics.Processing** as shown in the screenshot above. This displays the actual names of the physics engine tasks that run to update your Scene. The two most common names you're likely to see are:

Pxs: short for 'PhysX solver', which are physics engine tasks required by joints as well as resolving contacts for overlapping bodies.

ScScene: used for tasks required for updating the Scene, running the broad phase and narrow phase, and integrating bodies (moving them in space due to forces and impulses). See Steven M. LaValle's work on Planning Algorithms for a definition on two-phase **collision detection** phases.

GPU Profiler

Leave feedback

GPU Usage Profiler

The GPU Usage **Profiler** displays where GPU time is spent in your game. When you select this **Profiler**, the lower pane displays hierarchical time data for the selected frame. Select an item from the Hierarchy to see a breakdown of contributions in the right-hand panel. See documentation on the Profiler window to learn more about the information in the Profiler.

Note that GPU profiling is disabled when **Graphics Jobs (Experimental)** is enabled in the **Player Settings**. See documentation on Standalone Player Settings for more information. On macOS, GPU profiling is only available under OSX 10.9 Mavericks and later versions.

Remote profiling support

Platform	Graphics API	Status
Windows	D3D9, D3D11, D3D12, **OpenGL core**, OpenGL ES 2.0, OpenGL ES 3.x, Vulkan	Supported.
Mac OS X	OpenGL core	Supported.
	Metal	Not available. Use XCode's GPU Frame Debugger **UI** instead.
Linux	OpenGL core, Vulkan	Supported.
PlayStation 4	libgnm	Supported (an alternative is Razor).
Xbox One	D3D11	Supported (an alternative is PIX).
WebGL	**WebGL** 1.0 and **WebGL** 2.0	Not available.
Android	OpenGL ES 2.0, OpenGL ES 3.x	Supported only on devices running NVIDIA or Intel GPUs.
	Vulkan	Supported
iOS, tvOS	Metal, OpenGL ES 2.0, OpenGL ES 3.0	Not available. Use XCode's GPU Frame Debugger UI instead.
Tizen	OpenGL ES 2.0	Not available.

Profiling within the Unity Editor

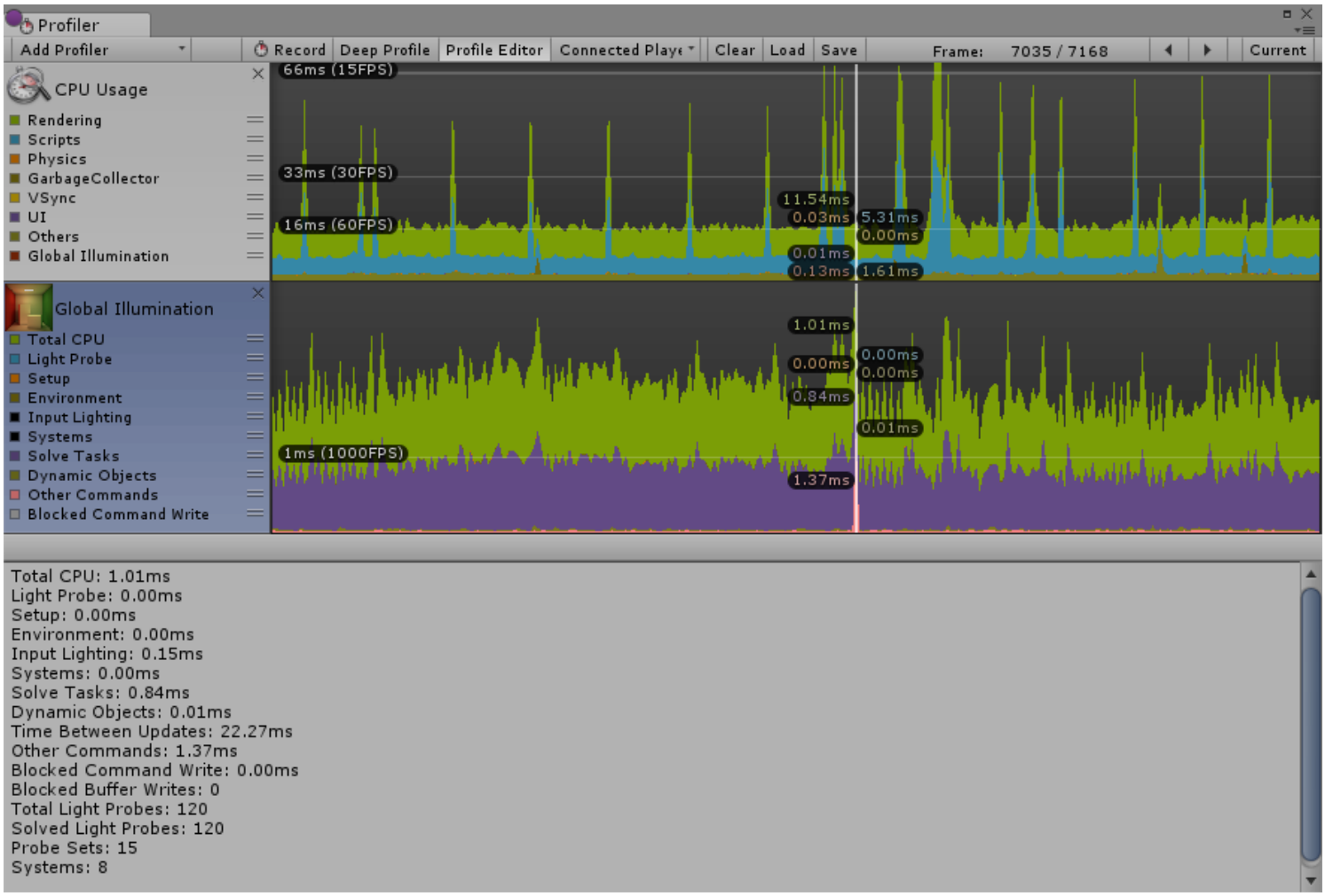
The Editor supports profiling on Windows with Direct3D 9 and Direct3D 11 APIs only. This has both advantages and disadvantages: it is convenient for quick profiling, because it means you don't need to build the Player; however, the Profiler is affected by the overhead of running the Unity Editor, which can make the profiling results less accurate.

2017–11–21 Page published with limited editorial review

Removed Samsung TV support.

Global Illumination Profiler

Leave feedback



The Global Illumination **Profiler** shows statistics and how much CPU time is consumed by the real-time Global Illumination (GI) subsystem across all worker threads. GI is managed in Unity by a piece of middleware called **Enlighten**. See documentation on Global Illumination for more information.

Name	Description
Total CPU Time	Total Enlighten CPU time across all threads.
Probe Update Time	Time spent updating **Light Probes**.
Setup Time	Time spent in the Setup stage.
Environment Time	Time spent processing Environment lighting.
Input Lighting Time	Time spent processing input lighting.
Systems Time	Time spent updating Systems.
Solve Tasks Time	Time spent running radiosity solver tasks.
Dynamic Objects Time	Time spent updating Dynamic **GameObjects**.
Time Between Updates	Time between Global Illumination updates.
Other Commands Time	Time spent processing other commands.
Blocked Command Write Time	Time spent in blocked state, waiting for command buffer.
Blocked Buffer Writes	Number of writes to the command buffer that were blocking.
Total Light Probes	Total number of Light Probes in the **Scene**.
Solved Light Probes	Number of solved Light Probes since the last update.
Probe Sets	Number of Light Probe sets in the Scene.
Systems	Number of Enlighten Systems in the Scene.

Pending Material GPU Renders	Number of Albedo/Emission renders queued for **rendering** on the GPU.
Pending Material Updates	Number of Material updates waiting to get processed.

2017–08–30 Page published with limited editorial review

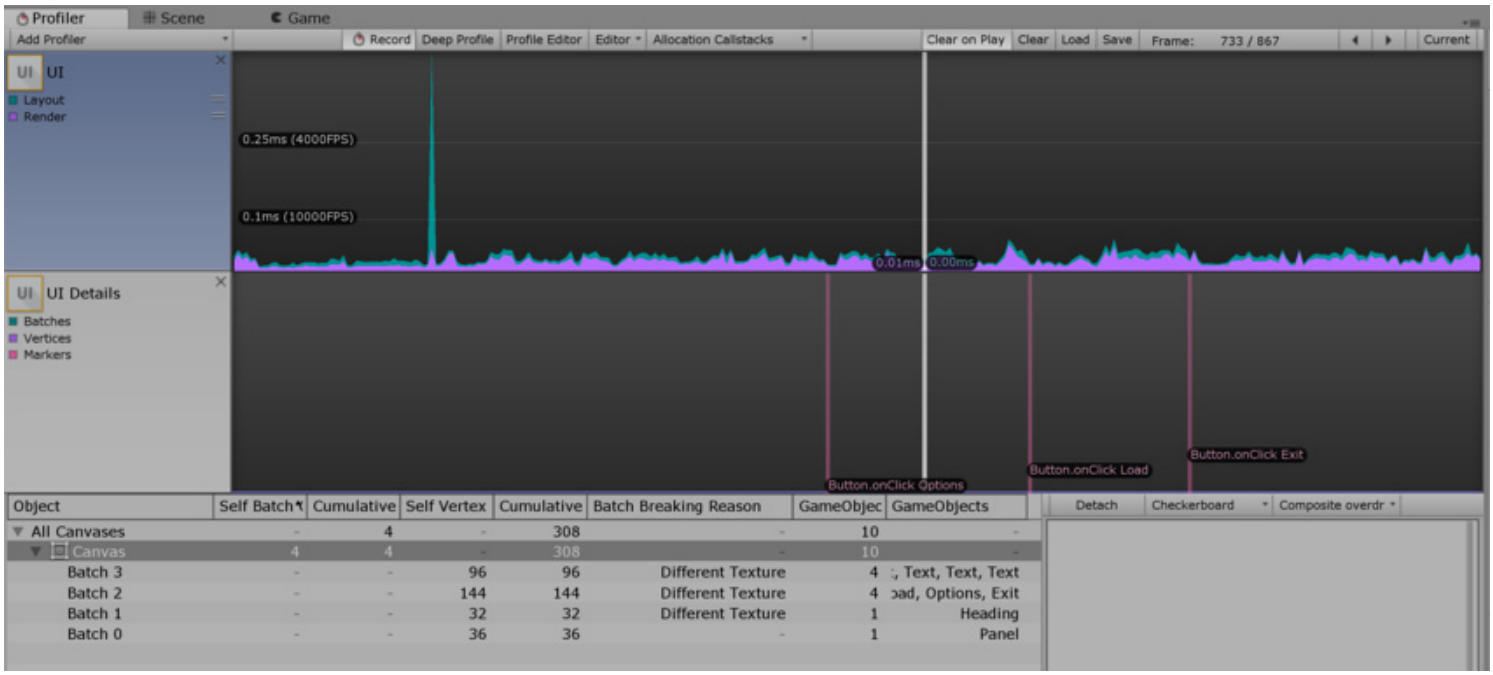
New feature in 2018.2

UI Profiler

Leave feedback

The **UI Profiler** is a **profiler** module dedicated to in-game **UI**.

Access it via the Profiler Window's menu: **Add Profiler** > **UI and UI Details**.



The UI and UI Details Profiler window

Use this feature to help with understanding the ui batching, why and how objects are batched, which part of the UI is responsible for a slow down, preview the UI or part of it when scrubbing the **timeline**.

Note that this Profiler is quite resource intensive, similar to other Profiler modules.

Settings

The UI Details chart has a togglable **Markers** group, similarl to what the CPU chart offers. In the preview panel, there's a button **Detach** and two drop-down menus.

The Markers toggle displays or hide event markers on the UI details chart.

Detach pops the preview out in a separate window.

The two drop-down menus allow you to choose the preview background (black, white, or checkerboard) and the preview type (original render, overdraw, or omposite overdraw).

Helpful Notes

Markers can be overwhelming, depending on the usecase profiled. Hiding or showing them when needed helps the chart readability.

To make visibility clearer, you can select the preview background according to the UI you are previewing. A white-ish UI on a white background won't be readable, for example, so you can change it.

Detaching the preview allows better screen estate management.

Overdraw and composite overdraw are used to determine which parts of the UI are drawn for nothing.

Definitions

Marker: markers are recorded when the user interacts with the UI (button click, slider value change, ...) and then drawn, if enabled, as vertical lines and labels on the chart.

Batch: the UI system tries to batch draw calls. There are many reasons two objects could not be batched together.

Batch Breaking Reasons

Not Coplanar With Canvas:
The batching needs the object's rect transform to be coplanar (unrotated) with the canvas.

CanvasInjectionIndex:
A CanvasGroup component is present and forces a new batch, ie. when displaying the drop down list of a combo box on top of the rest.

Different Material Instance, Rect clipping, Texture, A8TextureUsage:
Only objects with identical materials, masking, textures, texture alpha channel usage can be batched together.

Tips

Treeview rows have a context menu with a "find matching object in **scene**" entry, which is also triggerable by double clicking on a row.

2017–05–17 Page published with limited editorial review

New feature in Unity 2017.1

Multi-Scene editing

Leave feedback

Multi Scene Editing allows you to have multiple **scenes** open in the editor simultaneously, and makes it easier to manage **scenes** at runtime.

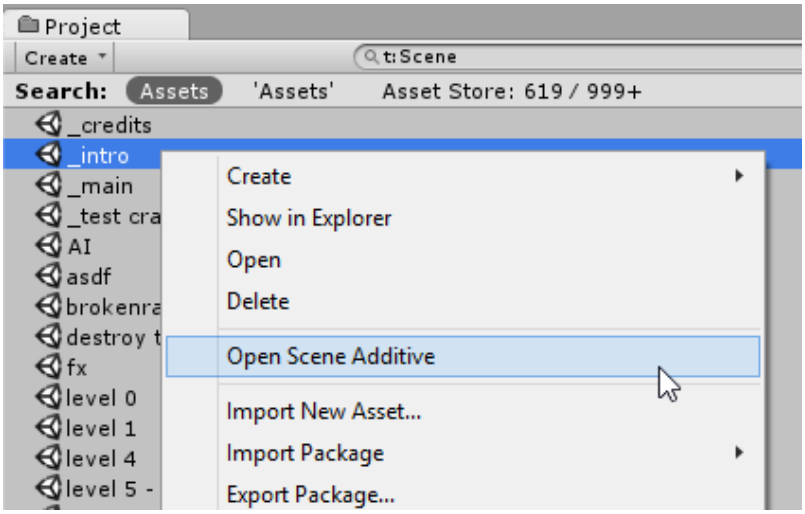
The ability to have multiple scenes open in the editor allows you to create large streaming worlds and improves the workflow when collaborating on scene editing.

This page describes:

The multi scene editing integration in the Editor
The Editor scripting and the Runtime scripting APIs
Current known issues

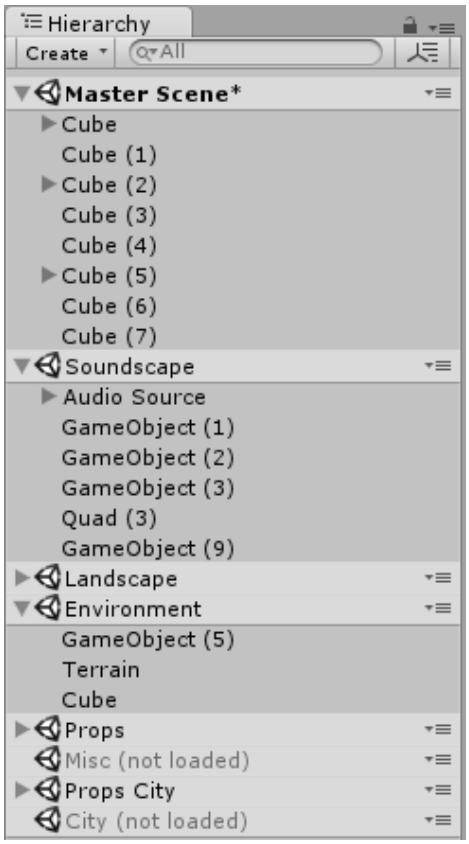
In the editor

To open a new scene and add it to the current list of scenes in the Hierarchy, either select *Open Scene Additive* in the context menu for a scene **asset**, or drag one or more scenes from the Project window into the Hierarchy Window.



Open Scene Additive will add the selected scene asset to the current scenes shown in the hierarchy

When you have multiple scenes open in the editor, each scene's contents are displayed separately in the hierarchy window. Each scene's contents appears below a scene divider bar which shows the scene's name and its save state.

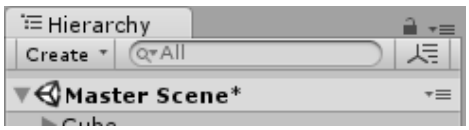


The Hierarchy window showing multiple scenes open simultaneously

While present in the hierarchy, scenes can be **loaded** or **unloaded** to reveal or hide the **gameobjects** contained within each scene. This is different to **adding** and **removing** them from the hierarchy window.

The scene dividers can be collapsed in the hierarchy to the scene's contents which may help you to navigate your hierarchy if you have lots of scenes loaded.

When working on multiple scenes, each scene that is modified will need its changes saved, so it is possible to have multiple unsaved scenes open at the same time. Scenes with unsaved changes will have an asterisk shown next to the name in the scene divider bar.

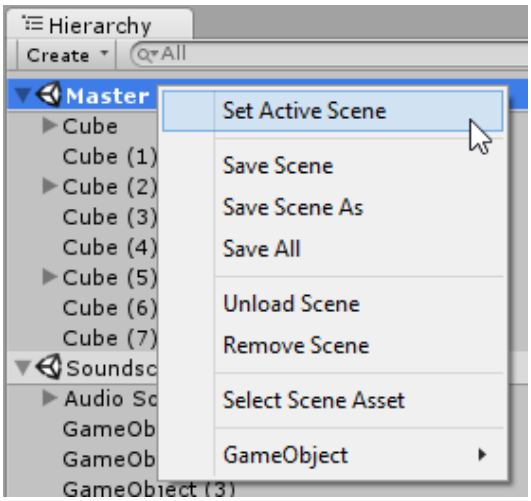


An asterisk in the scene divider indicating this scene has unsaved changes

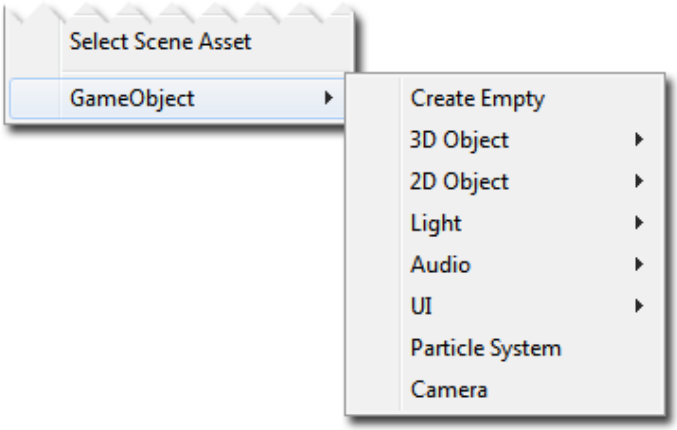
Each Scene can be saved separately via the context menu in the divider bar. Selecting "Save Scene" from the file menu or pressing Ctrl/Cmd + S will save changes to **all** open scenes.

The context menu in the scene divider bars allow you to perform other actions on the selected scene.

The Scene divider menu for loaded Scenes

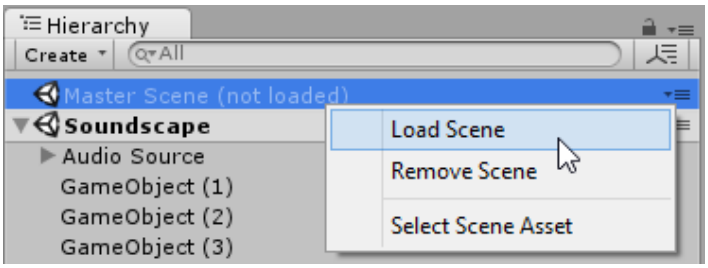


Set Active Scene	This allows you to specify which scene new Game Objects are created/instantiated in. There must always be one scene marked as the active scene
Save Scene	Saves the changes to the selected scene only.
Save Scene As	Saves the selected scene (along with any current modifications) as a new Scene asset.
Save All	Saves changes to all scenes.
Unload Scene	Unloads the scene, but keeps the scene in the Hierarchy window.
Remove Scene	Unloads and removes the scene from the Hierarchy window.
Select Scene Asset	Selects the scene's asset in the Project window.
GameObject	Provides a sub-menu allowing you to create GameObjects in the selected scene. The menu mirrors the creatable items available in Unity's main GameObject menu. (shown below)



The GameObject sub-menu in the Scene divider bar menu

The Scene divider menu for unloaded Scenes:



Load Scene	Loads the scene's contents

Remove Scene Remove the scene from the Hierarchy window.
Select Scene Asset Selects the scene's asset in the Project window.

Baking Lightmaps with multiple Scenes

To bake **Lightmap data** for multiple scenes at once, you should open the scenes that you want to bake, turn off "Auto" mode in the Lighting Window, and click the Build button to build the lighting.

The input to the lighting calculations is the static geometry and lights from all scenes. Therefore shadows and GI light bounces will work across all scenes. However, the lightmaps and realtime GI data are separated out into data that is loaded / unloaded separately for each scene. The lightmaps and realtime GI data atlases are split between scenes. This means lightmaps between scenes are never shared and they can be unloaded safely when unloading a scene. Lightprobe data is currently always shared and all lightprobes for all scenes baked together are loaded at the same time.

Alternatively, you can automate building lightmaps for multiple scenes by using the Lightmapping.BakeMultipleScenes function in an editor script.

Baking Navmesh data with multiple Scenes

To make **Navmesh data** for multiple scenes at once, you should open the scenes that you want to bake, and click the Bake button in the Navigation Window. The **navmesh** data will be baked into a single asset, shared by all loaded scenes. The data is saved into the folder matching the name of the current active scene (e.g. ActiveSceneName/NavMesh.asset). All loaded scenes will share this navmesh asset. After baking the navmesh, the scenes affected should be saved to make the scene-to-navmesh reference persistent.

Alternatively, you can automate building navmesh data for multiple scenes by using the NavMeshBuilder.BuildNavMeshForMultipleScenes function in an editor script.

Baking Occlusion Culling data with multiple Scenes

To bake **Occlusion Culling data** for multiple Scenes at once, open the Scenes that you want to bake, open the Occlusion Culling window (menu: **Window** > **Rendering** > **Occlusion Culling**) and click the **Bake** button. The Occlusion data is saved into *Library/Occlusion*, and a reference to the data is added in each open Scene. After baking the Occlusion Culling, save the Scenes affected to make the Scene-to-occlusion reference persistent.

Whenever a Scene is loaded additively, if it has the same occlusion data reference as the active Scene, the static renderers and portals culling information for that Scene are initialized from the occlusion data. Hereafter, the occlusion culling system performs as if all static renderers and portals were baked into a single Scene.

Play mode

In Play mode, with multiple scenes in the Hierarchy, an additional scene will show up called *DontDestroyOnLoad*.

Prior to Unity 5.3, any objects you would instantiate in Playmode marked as "DontDestroyOnLoad" would still show up in the hierarchy. These objects are not considered part of any scene but for Unity to still show the objects, and for you to inspect them, these objects are now shown as part of the special DontDestroyOnLoad scene.

You do not have access to the DontDestroyOnLoad scene and it is not available at runtime.

Scene-specific settings

A number of settings are specific to each scene. These are:

RenderSettings and LightmapSettings (both found in the Lighting Window)

NavMesh settings
Scene settings in the Occlusion Culling Window.
The way it works is that each scene will manage its own settings and only settings associated with that scene will be saved to the scene file.

If you have multiple scenes open, the settings that are used for rendering and navmesh are the ones from the **active scene**. This means that if you want to change the settings of a scene, you must either open only one scene and change the settings, or make the scene in question the active scene and change the settings.

When you switch active scene in the editor or at runtime, all the settings from the new scene will be applied and replace all previous settings.

Scripting

Editor scripting

For editor scripting we provide a Scene struct and EditorSceneManager API and a SceneSetup utility class.

The Scene struct is available both in the editor and at runtime and contains a handful of read-only properties relating to the scene itself, such as its name and asset path.

The EditorSceneManager class is only available in the editor. It is derived from SceneManager and has a number of functions that allow you to implement all the Multi Scene Editing features described above via editor scripting.

The SceneSetup class is a small utility class for storing information about a scene currently in the hierarchy.

The Undo and PrefabUtility classes have been extended to support multiple scenes. You can now instantiate a **prefab** in a given scene using [PrefabUtility.InstantiatePrefab], and you can move objects to the root of a scene in an un-doable manner using (Undo.MoveGameObjectToScene)[ScriptRef:Undo.MoveGameObjectToScene]

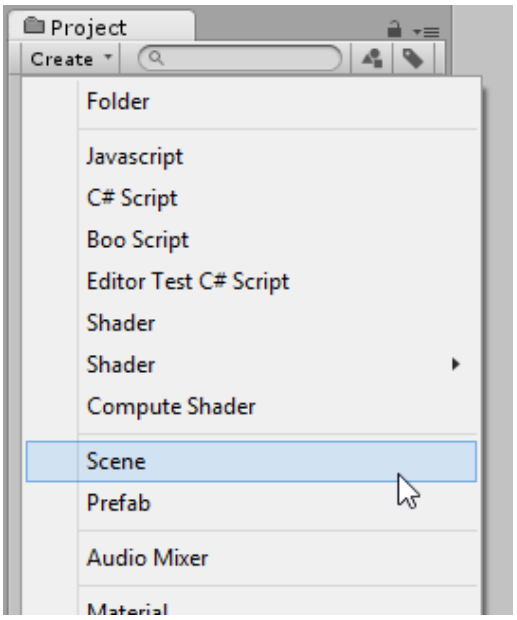
NOTE: *To use* `Undo.MoveGameObjectToScene`*, you must make sure the GameObject is already at the root of the scene it is currently in.*

Runtime scripting

For scripting at Runtime, the functions to work with multiple scenes such as **LoadScene** and **UnloadScene** are found on the SceneManager class.

Notes

In the File menu *Save Scene As* will only save the active scene. *Save Scene* will save all modified scenes, including prompting you to name the Untitled scene if it exists.



Creating a new Scene asset from the Project window's Create menu

Tips and tricks

It is possible to add a scene to the hierarchy while keeping it its *unloaded* state by holding Alt while dragging. This gives you the option to load the scene later, when desired.

New scenes can be created using the *Create* menu in the project window. New scenes will contain the default setup of Game Objects.

To avoid having to set up your hierarchy every time you restart unity or to make it easy to store different setups you can use EditorSceneManager.GetSceneManagerSetup to get a list of SceneSetup objects which describes the current setup. You can then serialize these into a ScriptableObject or something else along with any other information you might want to store about your scene setup. To restore the hierarchy simply recreate the list of SceneSetups and use EditorSceneManager.RestoreSceneManagerSetup.

At runtime to get the list of loaded scenes simply get sceneCount and iterate over the scenes using GetSceneAt.

You can get the scene a GameObject belongs to through GameObject.scene and you can move a GameObject to the root of a scene using SceneManager.MoveGameObjectToScene.

It is recommended to avoid using DontDestroyOnLoad to persist manager GameObjects that you want to survive across scene loads. Instead, create a **manager scene** that has all your managers and use SceneManager.LoadScene(*<path>*, LoadSceneMode.Additive) and SceneManager.UnloadScene to manage your game progress.

Known issues

Cross-Scene references are not supported, and are prevented in Edit mode. In Play mode they are allowed, because Scenes cannot be saved.

2017–10–24 Page amended with limited editorial review

Loading Resources at Runtime

Leave feedback

In some situations, it is useful to make an **asset** available to a project without loading it in as part of a **scene**. For example, there may be a character or other object that can appear in any **scene** of the game but which will only be used infrequently (this might be a "secret" feature, an error message or a highscore alert, say). Furthermore, you may even want to load assets from a separate file or URL to reduce initial download time or allow for interchangeable game content.

Unity supports **Resource Folders** in the project to allow content to be supplied in the main game file yet not be loaded until requested. You can also create **Asset Bundles**. These are files completely separate from the main game file which contain assets to be accessed by the game on demand from a file or URL.

Asset Bundles

An Asset Bundle is an external collection of assets. You can have many Asset Bundles and therefore many different external collections of assets. These files exist outside of the built Unity player, usually sitting on a web server for end-users to access dynamically.

To build an Asset Bundle, you call BuildPipeline.BuildAssetBundles() from inside an Editor script. In the arguments, you specify an array of **Objects** to be included in the built file, along with some other options. This will build a file that you can later load dynamically in the runtime by using AssetBundle.LoadAsset().

Resource Folders

Resource Folders are collections of assets that are included in the built Unity player, but are not necessarily linked to any **GameObject** in the **Inspector**.

To put anything into a Resource Folder, you simply create a new folder inside the **Project View**, and name the folder "Resources". You can have multiple Resource Folders organized differently in your Project. Whenever you want to load an asset from one of these folders, you call Resources.Load().

Note:

All assets found in the Resources folders and their dependencies are stored in a file called *resources.assets*. If an asset is already used by another level it is stored in the *.sharedAssets* file for that level. The **Edit -> PlayerSettings First Streamed Level** setting determines the level at which the *resources.assets* will be collected and included in the build.

If a level prior to *"First streamed Level"* is including an asset in a Resource folder, the asset will be stored in assets for that level. If it is included afterwards, the level will reference the asset from the "resources.assets" file.

Only assets that are in the *Resources folder* can be accessed through Resources.Load(). However many more assets might end up in the "resources.assets" file since they are dependencies. (For example a Material in the Resources folder might reference a Texture outside of the Resources folder)

Resource Unloading

You can unload resources of an AssetBundle by calling AssetBundle.Unload(). If you pass **true** for the **unloadAllLoadedObjects** parameter, both the objects held internally by the AssetBundle and the ones loaded from the AssetBundle using AssetBundle.LoadAsset() will be destroyed and memory used by the bundle will be released.

Sometimes you may prefer to load an AssetBundle, instantiate the objects desired and release the memory used up by the bundle while keeping the objects around. The benefit is that you free up memory for other tasks, for instance loading another AssetBundle. In this scenario you would pass **false** as the parameter. After the bundle is destroyed you will not be able to load objects from it any more.

If you want to destroy scene objects loaded using Resources.Load() prior to loading another level, call Object.Destroy() on them. To release assets, use Resources.UnloadUnusedAssets().

Plugins

Leave feedback

In Unity, you normally use **scripts** to create functionality, but you can also include code created outside Unity in the form of a **Plugin**. There are two kinds of plugins you can use in Unity: *Managed plugins* and *Native plugins*.

Managed plugins are managed .NET assemblies created with tools like Visual Studio. They contain only .NET code which means that they can't access any features that are not supported by the .NET libraries. However, managed code is accessible to the standard .NET tools that Unity uses to compile scripts. There is thus little difference in usage between **managed plugin** code and Unity script code, except for the fact that the plugins are compiled outside Unity and so the source may not be available.

Native plugins are platform-specific native code libraries. They can access features like OS calls and third-party code libraries that would otherwise not be available to Unity. However, these libraries are not accessible to Unity's tools in the way that managed libraries are. For example, if you forget to add a managed plugin file to the project, you will get standard compiler error messages. If you do the same with a **native plugin**, you will only see an error report when you try to run the project.

This section explains how to create plugins and use them in your Unity Projects.

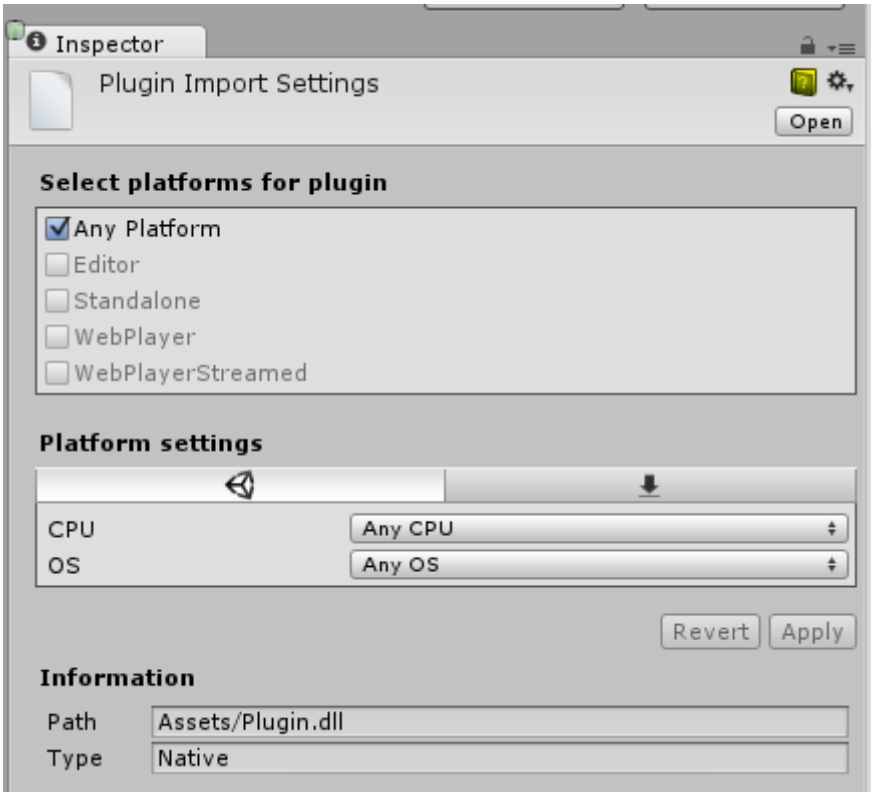
2018–03–19 Page amended with limited editorial review

MonoDevelop replaced by Visual Studio from 2018.1

Plugin Inspector

Leave feedback

The **Plugin Inspector** is used to select and manage target platforms for the plugins in your project. Simply select a **plugin** file to view its **Inspector**.



Inspector for a plugin called "CustomConnection"

Under **Select platforms for plugin**, choose which platforms will use the plugin by checking the appropriate boxes. If you select **Any Platform**, the plugin will apply to all platforms, including the Unity editor.

Once you have selected the platforms, you can choose additional options such as CPU type and specific OS from the separate **Platform Settings** section below. This contains a tab for each platform selected by the checkboxes. Most platforms have no settings, or just a few (such as CPU and OS selection).

The current list of file extensions that are treated as plugins and display the Plugin Inspector in the Unity Editor is found in `PluginImporter::CanLoadPathNameFile()`. The following file extensions identify files that are treated as plugins:

.dll
.winmd
.so
.jar
.aar
.xex
.def
.suprx
.prx
.sprx
.rpl

.cpp
.cc
.c
.h
.jslib
.jspre
.bc
.a
.m
.mm
.swift
.xib

Certain folders are treated as a single bundle plugin. No additional plugins are detected within such folders. The following extensions identify folders that are treated as bundle plugins:

.framework
.bundle
.plugin

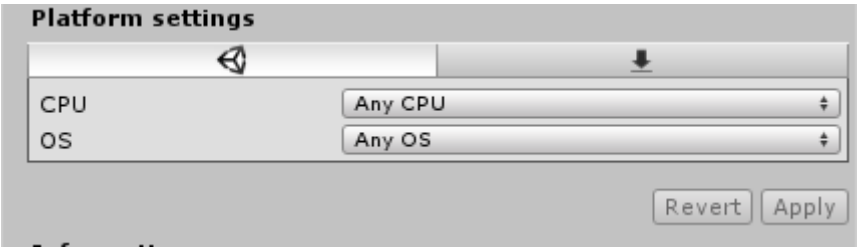
Default settings

To make transition easier from earlier Unity versions, Unity will try to set default plugins settings, depending on the folder where the plugin is located.

Folder	Default settings
Assets/../Editor	Plugin will be set only compatible with Editor, and won't be used when building to platform.
Assets/../Editor/(**x86** or **x86_64** or **x64**)	Plugin will be set only compatible with Editor, CPU value will be assigned depending on the subfolder.
Assets/../Plugins/(**x86_64** or **x64**)	x64 Standalone plugins will be set as compatible.
Assets/../Plugins/x86	x86 Standalone plugins will be set as compatible.
Assets/Plugins/Android/(**x86** or **armeabi** or **armeabi-v7a**)	Plugin will be set only compatible with Android, if CPU subfolder is present, CPU value will be set as well.
Assets/Plugins/iOS	Plugin will be set only compatible with **iOS**.
Assets/Plugins/WSA/(**x86** or **ARM**)	Plugin will be set only compatible with **Universal Windows Platform**, if subfolder is CPU present, CPU value will be set as well. Metro keyword can be used instead of WSA.
Assets/Plugins/WSA/(**SDK80** or **SDK81** or **PhoneSDK81**)	Same as above, additionally SDK value will be set, you can also add CPU subfolder afterwards. For compatibility reasons, SDK81 - Win81, PhoneSDK81 - WindowsPhone81.
Assets/Plugins/Tizen	Plugin will be set only compatible with Tizen.
Assets/Plugins/PS4	Plugin will be set only compatible with Playstation 4.

Device-specific settings

Editor settings



Options in the editor tab

For instance, if you select **CPU X86**, the plugin will be used only in **32 bit Editor**, but will not be used in **64 bit Editor**.

If you select **OS Windows**, the plugin will be used only in **Windows Editor**, but will not be used by the **OS X Editor**.

Standalone settings

See Standalone player settings.

Universal Windows Platform

See:

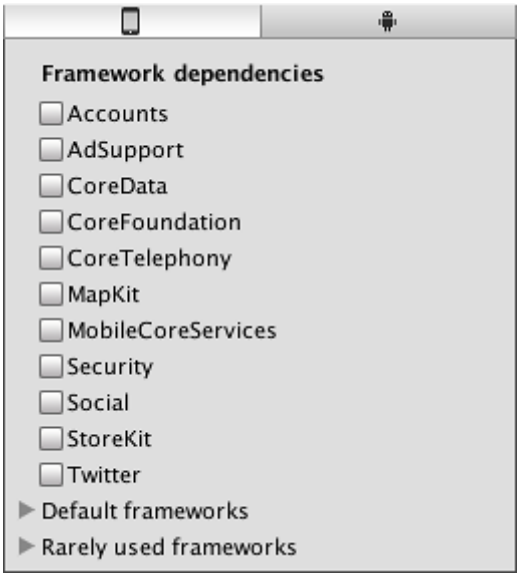
Universal Windows Platform: Plugins on .NET Scripting Backend

Universal Windows Platform: Plugins on IL2CPP Scripting Backend

Android

When building for Android, folders found with a parent path matching exactly **Assets/Plugins/Android/** are treated as an Android Library plugin folder. They are then treated in the same way as folders with the special extensions .plugin, .bundle and .framework.

iOS



iOS plugin settings, showing Framework dependencies

2017–11–20 Page amended with no editorial review

Removed Samsung TV support.

Managed Plugins

Leave feedback

Usually, **scripts** are kept in a project as source files and compiled by Unity whenever the source changes. However, it is also possible to compile a script to a **dynamically linked library** (DLL) using an external compiler. The resulting DLL can then be added to the project and the classes it contains can be attached to objects just like normal scripts.

It is generally much easier to work with scripts than DLLs in Unity. However, you may have access to third party Mono code which is supplied in the form of a DLL. When developing your own code, you may be able to use compilers not supported by Unity by compiling the code to a DLL and adding it to your Unity project. You may also want to supply Unity code without the source (for example, for an Asset Store product) and a DLL is an easy way to do this.

Creating a DLL

To create a DLL, you will first need a suitable compiler. Not all compilers that produce .NET code are guaranteed to work with Unity, so it may be wise to test the compiler with some available code before doing significant work with it. If the DLL contains no code that depends on the Unity API then you can simply compile it to a DLL using the appropriate compiler options. If you do want to use the Unity API then you will need to make Unity's own DLLs available to the compiler. On a Mac, these are contained in the application bundle (you can see the internal structure of the bundle by using the Show Package Contents command from the contextual menu; right click or ctrl-click the Unity application):-

The path to the Unity DLLs will typically be

```
/Applications/Unity/Unity.app/Contents/Managed/
```

...and the two DLLs are called UnityEngine.dll and UnityEditor.dll.

On Windows, the DLLs can be found in the folders that accompany the Unity application. The path will typically be

```
C:\Program Files\Unity\Editor\Data\Managed
```

...while the names of the DLLs are the same as for Mac OS.

The exact options for compiling the DLL will vary depending on the compiler used. As an example, the command line for the Mono C# compiler, **mcs** , might look like this on Mac OS:-

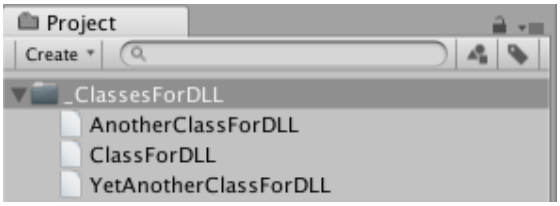
```
mcs -r:/Applications/Unity/Unity.app/Contents/Managed/UnityEngine.dll -target:library
```

Here, the *-r* option specifies a path to a library to be included in the build, in this case the UnityEngine library. The *-target* option specifies which type of build is required; the word "library" is used to select a DLL build. Finally, the name of the source file to compile is *ClassesForDLL.cs* (it is assumed that this file is in the current working folder, but you could specify

the file using a full path if necessary). Assuming all goes well, the resulting DLL file will appear shortly in the same folder as the source file.

Using the DLL

Once compiled, the DLL file can simply be dragged into the Unity project like any other **asset**. The DLL **asset** has a foldout triangle which can be used to reveal the separate classes inside the library. Classes that derive from MonoBehaviour can be dragged onto Game Objects like ordinary scripts. Non-MonoBehaviour classes can be used directly from other scripts in the usual way.



A folded-out DLL with the classes visible

Step by step guide for Visual Studio

This section explains how to build and integrate a simple DLL example with Visual Studio, and also how to prepare a debugging session for the DLL.

Setting Up the Project

First, open Visual Studio and create a new project. In Visual Studio, you should select **File > New > Project** and then choose **Visual C# > Class Library**.

You then need to fill out the information for the new library:

Name is the namespace (for this example use “DLLTest” as the name).
Location is the parent folder of the project.
Solution name is the folder of the project.
Next, you should add references to the Unity DLLs. In Visual Studio, open the contextual menu for *References* in the Solution Explorer and choose **Add Reference**. Then, choose the option **Browse > Browse > select file**.

At this stage, you will have the option to select the required DLL file. On Mac OS X, the file path is:

```
Applications/Unity.app/Contents/Managed/UnityEngine.dll
```

On Windows, the path is:

```
Program Files\Unity\Editor\Data\Managed\UnityEngine.dll
```

Code

For this example, rename the class to `MyUtilities` in the Solution browser and replace its code with the following:

```
using System;
using UnityEngine;

namespace DLLTest {

    public class MyUtilities {

        public int c;

        public void AddValues(int a, int b) {
            c = a + b;
        }

        public static int GenerateRandom(int min, int max) {
            System.Random rand = new System.Random();
            return rand.Next(min, max);
        }
    }
}
```

With the code in place, build the project to generate the DLL file along with its debug symbols.

Using the DLL in Unity

For this example, create a new project in Unity and copy the built file <project folder>/bin/Debug/DLLTest.dll into the Assets folder. Then, create a C# script called “Test” in Assets, and replace its contents with the following code:

```
using UnityEngine;
using System.Collections;
using DLLTest;

public class Test : MonoBehaviour {

     void Start () {
        MyUtilities utils = new MyUtilities();
        utils.AddValues(2, 3);
        print("2 + 3 = " + utils.c);
     }

     void Update () {
        print(MyUtilities.GenerateRandom(0, 100));
     }
}
```

When you attach this script to an object in the **scene** and press Play, you will see the output of the code from the DLL in the **Console window**.

Setting up a debugging session for the DLL

Firstly, you should prepare the debug symbols for the DLL. In Visual Studio, execute in the command prompt, passing <project folder>\bin\Debug\DLLTest.pdb as a parameter:

```
Program Files\Unity\Editor\Data\Mono\lib\mono\2.0\pdb2mdb.exe
```

Then, copy the converted file *<project folder>\bin\Debug\DLLTest.dll.mdb* into *Assets/Plugins*.

With this setup completed, you can debug code that uses the DLL in Unity in the usual way. See the Scripting Tools section for further information about debugging.

Compiling 'unsafe' C# code

You can enable support for compiling 'unsafe' C# code in Unity. To do this, go to **Edit** > **Project Settings** > **Player** and expand the **Other Settings** tab to reveal the **Allow 'unsafe' Code** checkbox.

2018–03–20 Page amended with limited editorial review

MonoDevelop replaced by Visual Studio from 2018.1

'unsafe C# Code checkbox' added in 2018.1

Native Plugins

Leave feedback

Unity has extensive support for native **Plugins**, which are libraries of native code written in C, C++, Objective-C, **etc**. Plugins allow your game code (written in Javascript or C#) to call functions from these libraries. This feature allows Unity to integrate with middleware libraries or existing C/C++ game code.

In order to use a **native plugin** you firstly need to write functions in a C-based language to access whatever features you need and compile them into a library. In Unity, you will also need to create a C# script which calls functions in the native library.

The native plugin should provide a simple C interface which the C# script then exposes to other user **scripts**. It is also possible for Unity to call functions exported by the native plugin when certain low-level **rendering** events happen (for example, when a graphics device is created), see the Native Plugin Interface page for details.

Example

A very simple native library with a single function might have source code that looks like this:

```
float FooPluginFunction () { return 5.0F; }
```

To access this code from within Unity, you could use code like the following:

```
using UnityEngine;
using System.Runtime.InteropServices;

class SomeScript : MonoBehaviour {

   #if UNITY_IPHONE

   // On iOS plugins are statically linked into
   // the executable, so we have to use __Internal as the
   // library name.
   [DllImport ("__Internal")]

   #else

   // Other platforms load plugins dynamically, so pass the name
   // of the plugin's dynamic library.
   [DllImport ("PluginName")]

   #endif

   private static extern float FooPluginFunction ();

   void Awake () {
      // Calls the FooPluginFunction inside the plugin
      // And prints 5 to the console
      print (FooPluginFunction ());
   }
}
```

Note that when using Javascript you will need to use the following syntax, where DLLName is the name of the plugin you have written, or "__Internal" if you are writing statically linked native code:

```
@DllImport (DLLName)
static private function FooPluginFunction () : float {};
```

Creating a Native Plugin

In general, plugins are built with native code compilers on the target platform. Since plugin functions use a C-based call interface, you must avoid name mangling issues when using C++ or Objective-C.

Further Information

Native Plugin Interface (this is needed if you want to do rendering in your plugin)
Mono Interop with native libraries
P-invoke documentation on MSDN

Building Plugins for Desktop Platforms

Leave feedback

This page describes Native Code Plugins for desktop platforms (Windows/Mac OS X/Linux).

Building a Plugin for Mac OS X

On Mac OSX, plugins are deployed as bundles. You can create the bundle project with XCode by selecting **File->NewProject...** and then selecting **Bundle** -> **Carbon/Cocoa Loadable Bundle** (in XCode 3) or **OS X** -> **Framework & Library** -> **Bundle** (in XCode 4)

If you are using C++ (.cpp) or Objective-C (.mm) to implement the plugin then you must ensure the functions are declared with C linkage to avoid name mangling issues.

```
extern "C" {
  float FooPluginFunction ();
}
```

Building a Plugin for Windows

Plugins on Windows are DLL files with exported functions. Practically any language or development environment that can create DLL files can be used to create plugins. As with Mac OSX, you should declare any C++ functions with C linkage to avoid name mangling issues.

Building a Plugin for Linux

Plugins on Linux are `.so` files with exported functions. These libraries are typically written in C or C++, but any language can be used. As with the other platforms, you should declare any C++ functions with C linkage in order to avoid name mangling issues.

32-bit and 64-bit libraries

The issue of needing 32-bit and/or 64-bit plugins is handled differently depending on the platform.

Windows and Linux

On Windows and Linux, plugins can be managed manually (e.g, before building a 64-bit player, you copy the 64-bit library into the `Assets/Plugins` folder, and before building a 32-bit player, you copy the 32-bit library into the `Assets/Plugins` folder) OR you can place the 32-bit version of the plugin in `Assets/Plugins/x86` and the 64-bit version of the plugin in `Assets/Plugins/x86_64`. By default the editor will look in the architecture-specific sub-directory first, and if that directory does not exist, it will copy plugins from the root `Assets/Plugins` folder instead.

Note that for the Universal Linux build, you are required to use the architecture-specific sub-directories (when building a Universal Linux build, the Editor will not copy any plugins from the root `Assets/Plugins` folder).

Mac OS X

For Mac OS X, you should build your plugin as a universal binary that contains both 32-bit and 64-bit architectures.

Using your plugin from C#

Once built, the bundle should be placed in the **Assets->Plugins** folder (or the appropriate architecture-specific sub-directory) in the Unity project. Unity will then find it by name when you define a function like this in the C# script:-

```
[DllImport ("PluginName")]
private static extern float FooPluginFunction ();
```

Please note that **PluginName** should not include the library prefix nor file extension. For example, the actual name of the plugin file would be PluginName.dll on Windows and libPluginName.so on Linux. Be aware that whenever you change code in the Plugin you will need to recompile **scripts** in your project or else the plugin will not have the latest compiled code.

Deployment

For cross platform plugins you must include the .bundle (for Mac), .dll (for Windows), and .so (for Linux) files in the Plugins folder. No further work is then required on your side - Unity automatically picks the right plugin for the target platform and includes it with the player.

Examples

Simplest Plugin

This plugin project implements only some very basic operations (print a number, print a string, add two floats, add two integers). This example may be helpful if this is your first Unity plugin. The project can be found here and includes Windows, Mac, and Linux project files.

Rendering from C++ code

An example multiplatform plugin that works with multithreaded **rendering** in Unity can be found on the Native Plugin Interface page.

Low-level Native Plugin Interface

Leave feedback

In addition to the basic script interface, Native Code Plugins in Unity can receive callbacks when certain events happen. This is mostly used to implement low-level **rendering** in your **plugin** and enable it to work with Unity's multithreaded **rendering**.

Headers defining interfaces exposed by Unity are provided with the editor.

Interface Registry

A plugin should export `UnityPluginLoad` and `UnityPluginUnload` functions to handle main Unity events. See `IUnityInterface.h` for the correct signatures. `IUnityInterfaces` is provided to the plugin to access further Unity APIs.

```
#include "IUnityInterface.h"
#include "IUnityGraphics.h"
// Unity plugin load event
extern "C" void UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API
    UnityPluginLoad(IUnityInterfaces* unityInterfaces)
{
    IUnityGraphics* graphics = unityInterfaces->Get<IUnityGraphics>();
}
```

Access to the Graphics Device

A plugin can access generic graphics device functionality by getting the `IUnityGraphics` interface. In earlier versions of Unity a `UnitySetGraphicsDevice` function had to be exported in order to receive notification about events on the graphics device. Starting with Unity 5.2 the new IUnityGraphics interface (found in `IUnityGraphics.h`) provides a way to register a callback.

```
#include "IUnityInterface.h"
#include "IUnityGraphics.h"

static IUnityInterfaces* s_UnityInterfaces = NULL;
static IUnityGraphics* s_Graphics = NULL;
static UnityGfxRenderer s_RendererType = kUnityGfxRendererNull;

// Unity plugin load event
extern "C" void UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API
    UnityPluginLoad(IUnityInterfaces* unityInterfaces)
{
    s_UnityInterfaces = unityInterfaces;
    s_Graphics = unityInterfaces->Get<IUnityGraphics>();
```

```
    s_Graphics->RegisterDeviceEventCallback(OnGraphicsDeviceEvent);

    // Run OnGraphicsDeviceEvent(initialize) manually on plugin load
    // to not miss the event in case the graphics device is already initialized
    OnGraphicsDeviceEvent(kUnityGfxDeviceEventInitialize);
}

// Unity plugin unload event
extern "C" void UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API
    UnityPluginUnload()
{
    s_Graphics->UnregisterDeviceEventCallback(OnGraphicsDeviceEvent);
}

static void UNITY_INTERFACE_API
    OnGraphicsDeviceEvent(UnityGfxDeviceEventType eventType)
{
    switch (eventType)
    {
        case kUnityGfxDeviceEventInitialize:
        {
            s_RendererType = s_Graphics->GetRenderer();
            //TODO: user initialization code
            break;
        }
        case kUnityGfxDeviceEventShutdown:
        {
            s_RendererType = kUnityGfxRendererNull;
            //TODO: user shutdown code
            break;
        }
        case kUnityGfxDeviceEventBeforeReset:
        {
            //TODO: user Direct3D 9 code
            break;
        }
        case kUnityGfxDeviceEventAfterReset:
        {
            //TODO: user Direct3D 9 code
            break;
        }
    };
}
```

Plugin Callbacks on the Rendering Thread

Rendering in Unity can be multithreaded if the platform and number of available CPUs will allow for it. When multithreaded rendering is used, the rendering API commands happen on a thread which is completely separate from the one that runs MonoBehaviour **scripts**. Consequently, it is not always possible for your plugin to start doing some rendering immediately, because it might interfere with whatever the render thread is doing at the time.

In order to do **any** rendering from the plugin, you should call GL.IssuePluginEvent from your script. This will cause the provided native function to be called from the render thread. For example, if you call GL.IssuePluginEvent from the **camera**'s OnPostRender function, you get a plugin callback immediately after the **camera** has finished rendering.

Signature for the `UnityRenderingEvent` callback is provided in `IUnityGraphics.h`. Native plugin code example:

```
// Plugin function to handle a specific rendering event
static void UNITY_INTERFACE_API OnRenderEvent(int eventID)
{
    //TODO: user rendering code
}

// Freely defined function to pass a callback to plugin-specific scripts
extern "C" UnityRenderingEvent UNITY_INTERFACE_EXPORT UNITY_INTERFACE_API
    GetRenderEventFunc()
{
    return OnRenderEvent;
}
```

Managed plugin code example:

```
#if UNITY_IPHONE && !UNITY_EDITOR
[DllImport ("__Internal")]
#else
[DllImport("RenderingPlugin")]
#endif
private static extern IntPtr GetRenderEventFunc();

// Queue a specific callback to be called on the render thread
GL.IssuePluginEvent(GetRenderEventFunc(), 1);
```

Such callbacks can now also be added to CommandBuffers via CommandBuffer.IssuePluginEvent.

Plugin using the OpenGL graphics API

There are two kind of OpenGL objects: Objects shared across OpenGL contexts (texture; buffer; renderbuffer; samplers; query; shader; and programs objects) and per-OpenGL context objects (vertex array; framebuffer; program pipeline; transform feedback; and sync objects).

Unity uses multiple OpenGL contexts. When initializing and closing the editor and the player, we rely on a **master** context but we use dedicated contexts for rendering. Hence, you can't create per-context objects during `kUnityGfxDeviceEventInitialize` and `kUnityGfxDeviceEventShutdown` events.

For example, a **native plugin** can't create a vertex array object during a `kUnityGfxDeviceEventInitialize` event and use it in a `UnityRenderingEvent` callback, because the active context is not the one used during the vertex array object creation.

Example

An example of a low-level rendering plugin is on bitbucket: **bitbucket.org/Unity-Technologies/graphicsdemos** (NativeRenderingPlugin folder). It demonstrates two things:

Renders a rotating triangle from C++ code after all regular rendering is done.
Fills a procedural texture from C++ code, using Texture.GetNativeTexturePtr to access it.
The project works with:

Windows (Visual Studio 2015) with Direct3D 9, Direct3D 11, Direct3D 12 and OpenGL.
Mac OS X (Xcode) with Metal and OpenGL.
Universal Windows Platform with Direct3D 11 and Direct3D 12.
WebGL

• 2017–05–16 Page amended with no editorial review

Low-level native plugin rendering extensions

Leave feedback

On top of the low-level native plugin interface, Unity also supports low level **rendering** extensions that can receive callbacks when certain events happen. This is mostly used to implement and control low-level **rendering** in your **plugin** and enable it to work with Unity's multithreaded **rendering**.

Due to the low-level nature of this extension the plugin might need to be preloaded before the devices get created. Currently the convention is name-based; the plugin name must begin *GfxPlugin* (for example: *GfxPluginMyNativePlugin*).

The rendering extension definition exposed by Unity is in the file *IUnityRenderingExtensions.h*, provided with the Editor (see file path *Unity\Editor\Data\PluginAPI*).

All platforms supporting **native plugins** support these extensions.

Rendering extensions API

To take advantage of the rendering extension, a plugin should export *UnityRenderingExtEvent* and optionally *UnityRenderingExtQuery*. There is a lot of documentation provided inside the include file.

Plugin callbacks on the rendering thread

A plugin gets called via *UnityRenderingExtEvent* whenever Unity triggers one of the built-in events. The callbacks can also be added to CommandBuffers via CommandBuffer.IssuePluginEventAndData or CommandBuffer.IssuePluginCustomBlit from **scripts**.

New feature in Unity 2017.1

2017–07–04 Page published with no editorial review

Low-level native plugin Shader compiler access

Leave feedback

On top of the low-level native plugin interface, Unity also supports low level access to the **shader** compiler, allowing the user to inject different variants into a **shader**. It is also an event driven approach in which the **plugin** will receive callbacks when certain builtin events happen.

The shader compiler access extension definition exposed by Unity is to be found in IUnityShaderCompilerAccess.h and it's provided with the editor.

These extensions are supported for now only on D3D11. Support for more platforms will follow.

Shader compiler access extension API

In order to take advantage of the **rendering** extension, a plugin should export UnityShaderCompilerExtEvent. There is a lot of documentation provided inside the include file.

A plugin will get called via UnityShaderCompilerExtEvent whenever one of the builtin events is triggered by Unity. The callbacks can also be added to CommandBuffers via CommandBuffer.IssuePluginEventAndData or CommandBuffer.IssuePluginCustomBlit command from **scripts**.

In addition to the basic script interface, Native Code Plugins in Unity can receive callbacks when certain events happen. This is mostly used to implement low-level rendering in your plugin and enable it to work with Unity's multithreaded rendering.

Headers defining interfaces exposed by Unity are provided with the editor.

Shader compiler access configuration interface

Unity provides an interface (IUnityShaderCompilerExtPluginConfigure) to which the shader compiler access is configured. This interface is used by the plugin to reserve its own keyword(s) and configure shader program and gpu program compiler masks (For what types for shader or GPU programs the plugin should be invoked)

New feature in Unity 2017.1

2017–07–04 Page published with no editorial review

AssetBundles

Leave feedback

An **AssetBundle** is an archive file containing platform specific **Assets** (Models, Textures, Prefabs, Audio clips, and even entire Scenes) that can be loaded at runtime. AssetBundles can express dependencies between each other; e.g. a material in AssetBundle A can reference a texture in AssetBundle B. For efficient delivery over networks, AssetBundles can be compressed with a choice of built-in algorithms depending on use case requirements (LZMA and LZ4).

AssetBundles can be useful for downloadable content (DLC), reducing initial install size, loading assets optimized for the end-user's platform, and reduce runtime memory pressure.

What's in an AssetBundle?

Good question, actually "AssetBundle" can refer to two different, but related things.

First is the actual file on disk. This we call the AssetBundle archive, or just archive for short in this document. The archive can be thought of as a container, like a folder, that holds additional files inside of it. These additional files consist of two types; the serialized file and resource files. The serialized file contains your assets broken out into their individual objects and written out to this single file. The resource files are just chunks of binary data stored separately for certain assets (textures and audio) to allow us to load them from disk on another thread efficiently.

Second is the actual AssetBundle object you interact with via code to load assets from a specific archive. This object contains a map of all the file paths of the assets you added to this archive to the objects that belong to that asset that need to be loaded when you ask for it.

2017–05–15 Page published with no editorial review

AssetBundle Workflow

Leave feedback

To get started with AssetBundles, follow these steps. More detailed information about each piece of the workflow can be found in the other pages in this section of documentation.

Assigning Assets to AssetBundles

To assign a given **Asset** to an AssetBundle, follow these steps:

Select the asset you want to assign to a bundle from your Project View
Examine the object in the inspector
At the bottom of the **inspector**, you should see a section to assign AssetBundles and Variants:
The left-hand drop down assigns the AssetBundle while the right-hand drop down assigns the variant
Click the left-hand drop down where it says “None” to reveal the currently registered AssetBundle names
Click “New...” to create a new AssetBundle
Type in the desired AssetBundle name. Note that AssetBundle names do support a type of folder structure depending on what you type. To add sub folders, separate folder names by a “/”. For example: AssetBundle name “environment/forest” will create a bundle named forest under an environment sub folder
Once you’ve selected or created an AssetBundle name, you can repeat this process for the right hand drop down to assign or create a Variant name, if you desire. Variant names are not required to build the AssetBundles
To read more information on AssetBundle assignments and accompanying strategies, see documentation on Preparing Assets for AssetBundles .

Build the AssetBundles

Create a folder called Editor in the Assets folders, and place a script with the following contents in the folder:

```
using UnityEditor;

public class CreateAssetBundles
{
    [MenuItem("Assets/Build AssetBundles")]
    static void BuildAllAssetBundles()
    {
        string assetBundleDirectory = "Assets/AssetBundles";
        if(!Directory.Exists(assetBundleDirectory))
{
    Directory.CreateDirectory(assetBundleDirectory);
}
BuildPipeline.BuildAssetBundles(assetBundleDirectory, BuildAssetBundleOptions.No
    }
}
```

This script will create a menu item at the bottom of the Assets menu called “Build AssetBundles” that will execute the code in the function associated with that tag. When you click Build AssetBundles a progress bar will appear with a build dialogue. This will take all the assets you labeled with an AssetBundle name in and place them in a folder at the path defined by assetBundleDirectory.

For more detail about what this code is doing, see documentation on Building AssetBundles.

Upload AssetBundles to off-site storage

This step is unique to each user and not a step Unity can tell you how to do. If you plan on uploading your AssetBundles to a third party hosting site, do that here. If you’re doing strictly local development and intend to have all of your AssetBundles on disk, skip to the next step.

Loading AssetBundles and Assets

For users intending to load from local storage, you’ll be interested in the AssetBundles.LoadFromFile API. Which looks like this:

```
public class LoadFromFileExample extends MonoBehaviour {
    function Start() {
        var myLoadedAssetBundle = AssetBundle.LoadFromFile(Path.Combine(Applicat
        if (myLoadedAssetBundle == null) {
            Debug.Log("Failed to load AssetBundle!");
            return;
        }
        var prefab = myLoadedAssetBundle.LoadAsset.<GameObject>("MyObject");
        Instantiate(prefab);
    }
}
```

`LoadFromFile` takes the path of the bundle file.

If you’re hosting your AssetBundles yourself and need to download them into your game, you’ll be interested in the UnityWebRequest API. Here’s an example:

```
IEnumerator InstantiateObject()

    {
        string uri = "file:///" + Application.dataPath + "/AssetBundles/" + asse
        UnityEngine.__Networking__.UnityWebRequest request = UnityEngine.__Netwo
        yield return request.Send();
```

```
        AssetBundle bundle = DownloadHandlerAssetBundle.GetContent(request);
        GameObject cube = bundle.LoadAsset<GameObject>("Cube");
        GameObject sprite = bundle.LoadAsset<GameObject>("Sprite");
        Instantiate(cube);
        Instantiate(sprite);
    }
```

`GetAssetBundle(string, int)` takes the uri of the location of the AssetBundle and the version of the bundle you want to download. In this example we're still pointing to a local file but the string uri could point to any url you have your AssetBundles hosted at.

The UnityWebRequest has a specific handle for dealing with AssetBundles, `DownloadHandlerAssetBundle`, which gets the AssetBundle from the request.

Regardless of the method you use, you'll now have access to the AssetBundle object. From that object you'll need to use `LoadAsset<T>(string)` which takes the type, `T`, of the asset you're attempting to load and the name of the object as a string that's inside the bundle. This will return whatever object you're loading from the AssetBundle. You can use these returned objects just like any object inside of Unity. For example, if you want to create a GameObject in the scene, you just need to call `Instantiate(gameObjectFromAssetBundle)`.

For more information on APIs that load AssetBundles, see documentation on Using AssetBundles Natively.

2017–05–15 Page published with no editorial review

Preparing Assets for AssetBundles

Leave feedback

When using AssetBundles you are free to assign any **asset** to any bundle you desire. However, there are certain strategies to consider when setting up your bundles. These grouping strategies are meant to be used however you see fit for your specific project. Feel free to mix and match these strategies as you see fit.

Logical Entity Grouping

Logical Entity Grouping is where assets are assigned to AssetBundles based on the functional portion of the project they represent. This includes sections such as User-Interface, characters, environments, and anything else that may appear frequently throughout the lifetime of the application.

Examples

Bundling all the textures and layout data for a User-Interface screen
Bundling all the models and animations for a character/set of characters
Bundling the textures and models for pieces of the scenery shared across multiple levels
Logical Entity Grouping is ideal for downloadable content (DLC) for the fact that, with everything separated in this way, you're able to make a change to a single entity and not require the download of additional, unchanged, assets.

The biggest trick to being able to properly implement this strategy is that the developer assigning assets to their respective bundles must be familiar with precisely when and where each asset will be used by the project.

Type Grouping

For this strategy you'll assign assets that are of similar type, such as audio tracks or language localization files, to a single AssetBundle.

Type grouping is one of the better strategies for building AssetBundles to be used by multiple platforms. For example if your audio **compression** settings are identical between windows and mac platforms, you can pack all audio data into AssetBundles by themselves and reuse those bundles, whereas shaders tend to get compiled with more platform specific options so a **shader** bundle you build for mac may not be reused on windows. In addition, this method is great for making your AssetBundles compatible with more unity player versions as textures **compression** formats and settings change less frequently than something like your code **scripts** or **prefabs**.

Concurrent Content Grouping

Concurrent Content Grouping is the idea that you will bundle assets together that will be loaded and used at the same time. You could think of these types of bundles as being used for a level based game where each level contains totally unique characters, textures, music, **etc**. You would want to be absolutely certain that an asset in one of these AssetBundles is only used at the same time the rest of the assets in that bundle are going to be used. Having a dependency on a single asset inside a Concurrent Content Grouping bundle would result in significant increased load times. You would be forced to download the entire bundle for that single asset.

The most common use-case for Concurrent Content Grouping bundles is for bundles that are based on **scenes**. In this assignment strategy, each scene bundle should contain most or all of that **scenes** dependencies.

Note that a project absolutely can and should mix these strategies as your needs require. Using the optimal asset assignment strategy for any given scenario greatly increases efficiency for any project.

For example, a project may decide to group its User-Interface (UI) elements for different platforms into their own Platform-UI specific bundle but group its interactive content by level/scene.

Regardless of the strategy you follow, here are some additional tips that are good to keep in mind across the board:

Split frequently updated objects into AssetBundles separate from objects that rarely change
Group objects that are likely to be loaded simultaneously. Such as a model, its textures, and its animations

If you notice multiple objects across multiple AssetBundles are dependant on a single asset from a completely different AssetBundle, move the dependency to a separate AssetBundle. If several AssetBundles are referencing the same group of assets in other AssetBundles, it may be worth pulling those dependencies into a shared AssetBundle to reduce duplication.
If two sets of objects are unlikely to ever be loaded at the same time, such as Standard and High Definition assets, be sure they are in their own AssetBundles.
Consider splitting apart an AssetBundle if less that 50% of that bundle is ever frequently loaded at the same time
Consider combining AssetBundles that are small (less that 5 to 10 assets) but whose content is frequently loaded simultaneously
If a group of objects are simply different versions of the same object, consider AssetBundle Variants

---

2017–05–15 Page published with no editorial review

Building AssetBundles

Leave feedback

In the documentation on the AssetBundle Workflow, we have a code sample which passes three arguments to the `BuildPipeline.BuildAssetBundles` function. Let's dive a little deeper into what we're actually saying.

Assets/AssetBundles: This is the directory that the AssetBundles will be output to. You can change this to any output directory you desire, just ensure that the folder actually exists before you attempt a build.

BuildAssetBundleOptions

There are several different `BuildAssetBundleOptions` that you can specify that have a variety of effects. See Scripting API Reference on BuildAssetBundleOptions for a table of all the options.

While you're free to combine `BuildAssetBundleOptions` as needs change and arise, there are three specific `BuildAssetBundleOptions` that deal with AssetBundle **Compression**:

`BuildAssetBundleOptions.None`: This bundle option uses LZMA Format compression, which is a single compressed LZMA stream of serialized data files. LZMA compression requires that the entire bundle is decompressed before it's used. This results in the smallest possible file size but a slightly longer load time due to the decompression. It is worth noting that when using this BuildAssetBundleOptions, in order to use any assets from the bundle the entire bundle must be uncompressed initially.
Once the bundle has been decompressed, it will be recompressed on disk using LZ4 compression which doesn't require the entire bundle be decompressed before using assets from the bundle. This is best used when a bundle contains assets such that to use one asset from the bundle would mean all assets are going to be loaded. Packaging all assets for a character or scene are some examples of bundles that might use this.
Using LZMA compression is only recommended for the initial download of an AssetBundle from an off-site host due to the smaller file size. Once the file has been downloaded, it will be cached as a LZ4 compressed bundle.

`BuildAssetBundleOptions.UncompressedAssetBundle`: This bundle option builds the bundles in such a way that the data is completely uncompressed. The downside to being uncompressed is the larger file download size. However, the load times once downloaded will be much faster.

`BuildAssetBundleOptions.ChunkBasedCompression`: This bundle option uses a compression method known as LZ4, which results in larger compressed file sizes than LZMA but does not require that entire bundle is decompressed, unlike LZMA, before it can be used. LZ4 uses a chunk based algorithm which allows the AssetBundle be loaded in pieces or "chunks." Decompressing a single chunk allows the contained assets to be used even if the other chunks of the AssetBundle are not decompressed.

Using `ChunkBasedCompression` has comparable loading times to uncompressed bundles with the added benefit of reduced size on disk.

BuildTarget

`BuildTarget.Standalone`: Here we're telling the build pipeline which target platform we are going to be using these AssetBundles for. You can find a list of the available explicit build targets in the Scripting API Reference for BuildTarget. However, if you'd rather not hardcode in your build target, feel free to take advantage of `EditorUserBuildSettings.activeBuildTarget` which will automatically find the platform you're currently setup to build for and build your AssetBundles based on that target.

Once you've properly set up your build script, it's finally time to build your bundles. If you followed the script example above, click **Assets** > **Build AssetBundles** to kick off the process.

Now that you've successfully built your AssetBundles, you may notice that your AssetBundles directory has more files than you might have originally expected. 2*(n+1) more files, to be exact. Let's take a minute and go over exactly what the `BuildPipeline.BuildAssetBundles` yields.

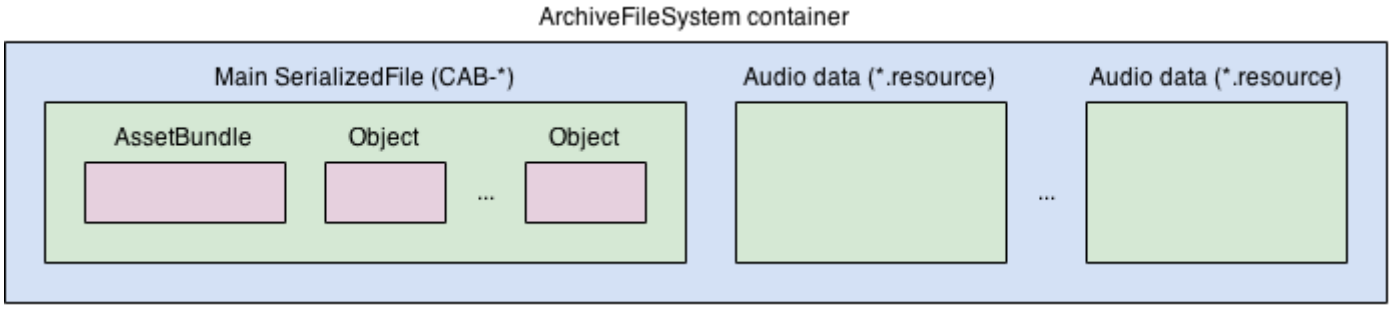
For every AssetBundle you specified in the editor, you'll notice a file with your AssetBundle name and your AssetBundle name + ".manifest".

There will be an additional bundle and manifest that doesn't share a name with any AssetBundle you created. It, instead, is named after the directory that it's located in (where the AssetBundles were built to). This is the Manifest Bundle. We'll discuss more about this and how to use it in the future.

The AssetBundle File

This is the file that lacks the .manifest extension and what you'll be loading in at runtime in order to load your Assets.

The AssetBundle file is an archive that contains multiple files internally. The structure of this archive can change slightly depending on if it is an AssetBundle or a Scene AssetBundle. This is the structure of a normal AssetBundle:



The Scene AssetBundle is different to normal AssetBundles, in that it is optimized for stream loading of a Scene and its content.

The Manifest File

For every bundle generated, including the additional Manifest Bundle, an associated manifest file is generated. The manifest file can be opened with any text editor and contains information such as the cyclic redundancy check (CRC) data and dependency data for the bundle. For the normal AssetBundles their manifest file will look something like this:

```
ManifestFileVersion: 0
CRC: 2422268106
Hashes:
  AssetFileHash:
    serializedVersion: 2
    Hash: 8b6db55a2344f068cf8a9be0a662ba15
  TypeTreeHash:
    serializedVersion: 2
```

```
    Hash: 37ad974993dbaa77485dd2a0c38f347a
HashAppended: 0
ClassTypes:
- Class: 91
  Script: {instanceID: 0}
Assets:
  Asset_0: Assets/Mecanim/StateMachine.controller
Dependencies: {}
```

Which shows the contained assets, dependencies, and other information.

The Manifest Bundle that was generated will have a manifest, but it'll look more like this:

```
ManifestFileVersion: 0
AssetBundleManifest:
  AssetBundleInfos:
    Info_0:
      Name: scene1assetbundle
      Dependencies: {}
```

This will show how AssetBundles relate and what their dependencies are. For now, just understand that this bundle contains the AssetBundleManifest object which will be incredibly useful for figuring out which bundle dependencies to load at runtime. To learn more about how to use this bundle and the manifest object, see documentation on Using AssetBundles Natively.

• 2017–05–15 Page published with no editorial review

AssetBundle Dependencies

Leave feedback

AssetBundles can become dependent on other AssetBundles if one or more of the `UnityEngine.Objects` contains a reference to a `UnityEngine.Object` located in another bundle. A dependency does not occur if the `UnityEngine.Object` contains a reference to a `UnityEngine.Object` that is not contained in any AssetBundle. In this case, a copy of the object that the bundle would be dependent on is copied into the bundle when you build the AssetBundles. If multiple objects in multiple bundles contain a reference to the same object that isn't assigned to a bundle, every bundle that would have a dependency on that object will make its own copy of the object and package it into the built AssetBundle.

Should an AssetBundle contain a dependency, it is important that the bundles that contain those dependencies are loaded before the object you're attempting to instantiate is loaded. Unity will not attempt to automatically load dependencies.

Consider the following example, a Material in **Bundle 1** references a Texture in **Bundle 2**:

In this example, before loading the Material from **Bundle 1**, you would need to load **Bundle 2** into memory. It does not matter which order you load **Bundle 1** and **Bundle 2**, the important takeaway is that **Bundle 2** is loaded before loading the Material from **Bundle 1**. In the next section, we'll discuss how you can use the `AssetBundleManifest` objects we touched on in the previous section to determine, and load, dependencies at runtime.

2017–05–15 Page published with no editorial review

Using AssetBundles Natively

Leave feedback

There are four different APIs that you can use to load AssetBundles. Their behavior varies based on the platform the bundle is being loaded and the **compression** method used when the AssetBundles were built (uncompressed, LZMA, LZ4).

The four APIs we have to work with are:

AssetBundle.LoadFromMemoryAsync
AssetBundle.LoadFromFile
WWW.LoadfromCacheOrDownload
UnityWebRequest's DownloadHandlerAssetBundle (Unity 5.3 or newer)

AssetBundle.LoadFromMemoryAsync

AssetBundle.LoadFromMemoryAsync

This function takes an array of bytes that contains AssetBundle data. Optionally you can also pass in a CRC value if you desire. If the bundle is LZMA compressed it will decompress the AssetBundle while it's loading. LZ4 compressed bundles are loaded in their compressed state.

Here's one example of how to use this method:

```
using UnityEngine;
using System.Collections;
using System.IO;

public class Example : MonoBehaviour
{
    IEnumerator LoadFromMemoryAsync(string path)
    {
        AssetBundleCreateRequest createRequest = AssetBundle.LoadFromMemoryAsync
        yield return createRequest;
        AssetBundle bundle = createRequest.assetBundle;
        var prefab = bundle.LoadAsset<GameObject>("MyObject");
        Instantiate(prefab);
    }
}
```

However, this is not the only strategy that makes using LoadFromMemoryAsync possible. File.ReadAllBytes(path) could be replaced with any desired procedure of obtaining a byte array.

AssetBundle.LoadFromFile

AssetBundle.LoadFromFile

This API is highly-efficient when loading uncompressed bundles from local storage. LoadFromFile will load the bundle directly from disk if the bundle is uncompressed or chunk (LZ4) compressed. Loading a fully compressed (LZMA) bundle with this method will first decompress the bundle before loading it into memory.

One example of how to use `LoadFromFile`:

```
public class LoadFromFileExample extends MonoBehaviour {
    function Start() {
        var myLoadedAssetBundle = AssetBundle.LoadFromFile(Path.Combine(Applicat
        if (myLoadedAssetBundle == null) {
            Debug.Log("Failed to load AssetBundle!");
            return;
        }
        var prefab = myLoadedAssetBundle.LoadAsset.<GameObject>("MyObject");
        Instantiate(prefab);
    }
}
```

Note: On Android devices with Unity 5.3 or older, this API will fail when trying to load AssetBundles from the Streaming **Assets** path. This is because the contents of that path will reside inside a compressed .jar file. Unity 5.4 and newer can use this API call with Streaming **Assets** just fine.

WWW.LoadFromCacheOrDownload

WWW.LoadFromCacheOrDownload

TO BE DEPRECATED (Use UnityWebRequest)

This API is useful for downloading AssetBundles from a remote server or loading local AssetBundles. It is the older, and less desirable version of the UnityWebRequest API.

Loading an AssetBundle from a remote location will automatically cache the AssetBundle. If the AssetBundle is compressed, a worker thread will spin up to decompress the bundle and write it to the cache. Once a bundle has been decompressed and cached, it will load exactly like AssetBundle.LoadFromFile.

One example of how to use `LoadFromCacheOrDownload`:

```
using UnityEngine;
using System.Collections;

public class LoadFromCacheOrDownloadExample : MonoBehaviour
{
    IEnumerator Start ()
    {
```

```
            while (!Caching.ready)
                    yield return null;

        var www = WWW.LoadFromCacheOrDownload("http://myserver.com/myassetBundle
        yield return www;
        if(!string.IsNullOrEmpty(www.error))
        {
            Debug.Log(www.error);
            yield return;
        }
        var myLoadedAssetBundle = www.assetBundle;

        var asset = myLoadedAssetBundle.mainAsset;
    }
}
```

Due to the memory overhead of caching an AssetBundle's bytes in the WWW object, it is recommended that all developers using WWW.LoadFromCacheOrDownload ensure that their AssetBundles remain small - a few megabytes, at most. It is also recommended that developers operating on limited-memory platforms, such as mobile devices, ensure that their code downloads only a single AssetBundle at a time, in order to avoid memory spikes.

If the cache folder does not have any space for caching additional files, LoadFromCacheOrDownload will iteratively delete the least-recently-used AssetBundles from the Cache until sufficient space is available to store the new AssetBundle. If making space is not possible (because the hard disk is full, or all files in the cache are currently in use), LoadFromCacheOrDownload() will bypass Caching and stream the file into memory

In order to force LoadFromCacheOrDownload the version parameter (the second parameter) will need to change. The AssetBundle will only be loaded from cache if the version passed to the function matches the version of the currently cached AssetBundle.

UnityWebRequest

UnityWebRequest

The UnityWebRequest has a specific API call to deal with AssetBundles. To begin, you'll need to create your web request using `UnityWebRequest.GetAssetBundle`. After returning the request, pass the request object into `DownloadHandlerAssetBundle.GetContent(UnityWebRequest)`. This `GetContent` call will return your AssetBundle object.

You can also use the `assetBundle` property on the DownloadHandlerAssetBundle class after downloading the bundle to load the AssetBundle with the efficiency of `AssetBundle.LoadFromFile`.

Here's an example of how to load an AssetBundle that contains two **GameObjects** and Instantiate them. To begin this process, we'd just need to call `StartCoroutine(InstantiateObject());`

```
IEnumerator InstantiateObject()

    {
        string uri = "file:///" + Application.dataPath + "/AssetBundles/" + asse
        yield return request.Send();
        AssetBundle bundle = DownloadHandlerAssetBundle.GetContent(request);
        GameObject cube = bundle.LoadAsset<GameObject>("Cube");
        GameObject sprite = bundle.LoadAsset<GameObject>("Sprite");
        Instantiate(cube);
        Instantiate(sprite);
    }
```

The advantages of using UnityWebRequest is that it allows developers to handle the downloaded data in a more flexible manner and potentially eliminate unnecessary memory usage. This is the more current and preferred API over the UnityEngine.WWW class.

Loading Assets from AssetBundles

Now that you've successfully downloaded your AssetBundle, it's time to finally load in some Assets.

Generic code snippet:

```
T objectFromBundle = bundleObject.LoadAsset<T>(assetName);
```

T is the type of the Asset you're attempting to load.

There are a couple options when deciding how to load Assets. We have `LoadAsset`, `LoadAllAssets`, and their Async counterparts `LoadAssetAsync` and `LoadAllAssetsAsync` respectively.

This is how to load an asset from an AssetBundles synchronously:

To load a single GameObject:

```
GameObject gameObject = loadedAssetBundle.LoadAsset<GameObject>(assetName);
```

To load all Assets:

```
Unity.Object[] objectArray = loadedAssetBundle.LoadAllAssets();
```

Now, where as the previously shown methods return either the type of object you're loading or an array of objects, the asynchronous methods return an AssetBundleRequest. You'll need to wait for this operation to complete before accessing the asset. To load an asset:

```
AssetBundleRequest request = loadedAssetBundleObject.LoadAssetAsync<GameObject>(
yield return request;
var loadedAsset = request.asset;
```

And

```
AssetBundleRequest request = loadedAssetBundle.LoadAllAssetsAsync();
yield return request;
var loadedAssets = request.allAssets;
```

Once you have loaded your Assets you're good to go! You're able to use the loaded objects as you would any Object in Unity.

Loading AssetBundle Manifests

Loading AssetBundle manifests can be incredibly useful. Especially when dealing with AssetBundle dependencies.

To get a useable AssetBundleManifest object, you'll need to load that additional AssetBundle (the one that's named the same thing as the folder it's in) and load an object of type AssetBundleManifest from it.

Loading the manifest itself is done exactly the same as any other Asset from an AssetBundle:

```
AssetBundle assetBundle = AssetBundle.LoadFromFile(manifestFilePath);
AssetBundleManifest manifest = assetBundle.LoadAsset<AssetBundleManifest>("Asset
```

Now you have access to the `AssetBundleManifest` API calls through the manifest object from the above example. From here you can use the manifest to get information about the AssetBundles you built. This information includes dependency data, hash data, and variant data for the AssetBundles.

Remember in the earlier section when we discussed AssetBundle Dependencies and how, if a bundle had a dependency on another bundle, those bundles would need to be loaded in before loading any Assets from the

original bundle? The manifest object makes dynamically finding a loading dependencies possible. Let's say we want to load all the dependencies for an AssetBundle named "assetBundle".

```
AssetBundle assetBundle = AssetBundle.LoadFromFile(manifestFilePath);
AssetBundleManifest manifest = assetBundle.LoadAsset<AssetBundleManifest>("Asset
string[] dependencies = manifest.GetAllDependencies("assetBundle"); //Pass the n
foreach(string dependency in dependencies)
{
    AssetBundle.LoadFromFile(Path.Combine(assetBundlePath, dependency));
}
```

Now that you're loading AssetBundles, AssetBundle dependencies, and Assets, it's time to talk about managing all of these loaded AssetBundles.

Managing Loaded AssetBundles

See also: Unity Learn tutorial on Managing Loaded AssetBundles

Unity does not automatically unload Objects when they are removed from the active **scene**. Asset cleanup is triggered at specific times, and it can also be triggered manually.

It is important to know when to load and unload an AssetBundle. Improperly unloading an AssetBundle can lead to duplicating objects in memory or other undesirable circumstances, such as missing textures.

The biggest thing to understand about AssetBundle management is when to call

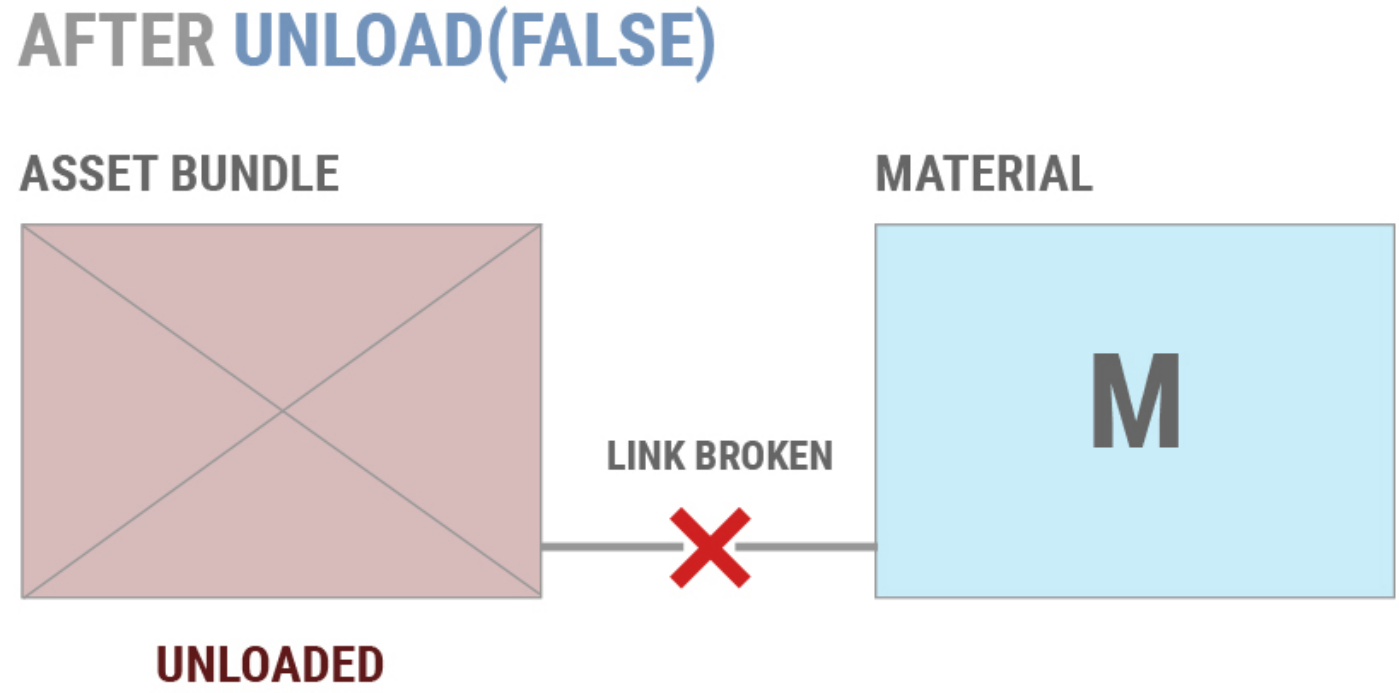
AssetBundle.Unload(bool); and if you should pass true or false into the function call. Unload is a non-static function that will unload your AssetBundle. This API unloads the header information of the AssetBundle being called. The argument indicates whether to also unload all Objects instantiated from this AssetBundle.

`AssetBundle.Unload(true)` unloads all GameObjects (and their dependencies) that were loaded from the AssetBundle. This does not include copied GameObjects (such as Instantiated GameObjects), because they no longer belong to the AssetBundle. When this happens, Textures that are loaded from that AssetBundle (and still belong to it) disappear from GameObjects in the Scene, and Unity treats them as missing Textures.

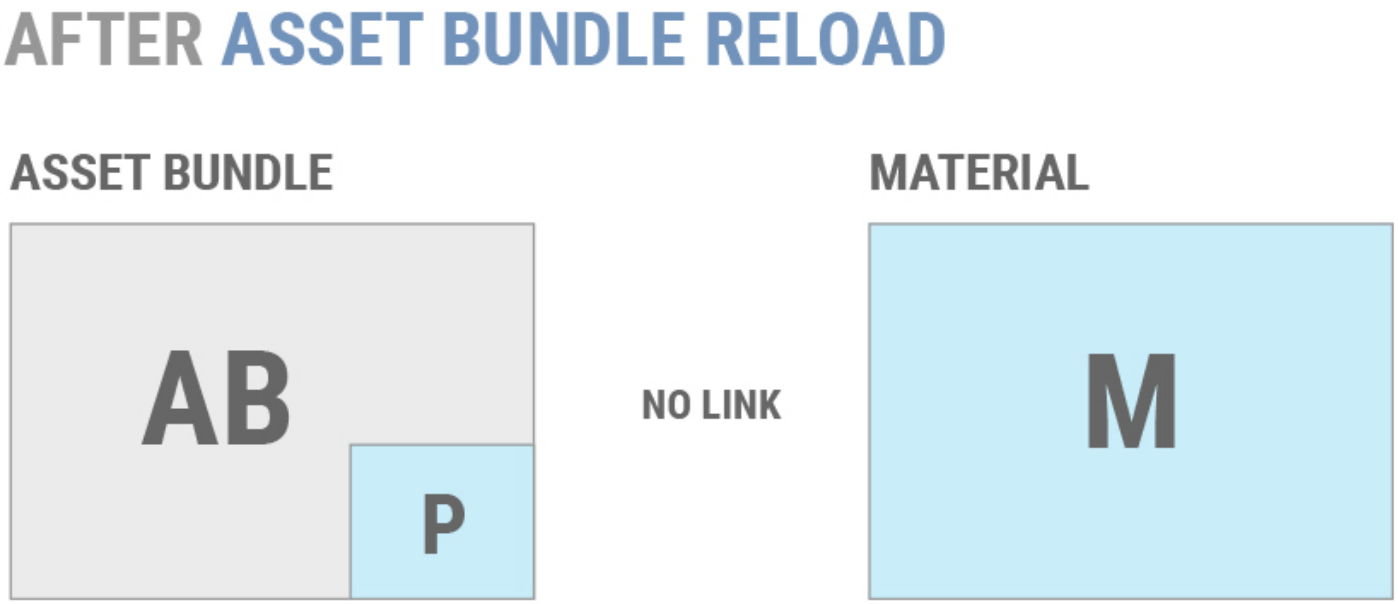
Let's assume Material M is loaded from AssetBundle AB as shown below.

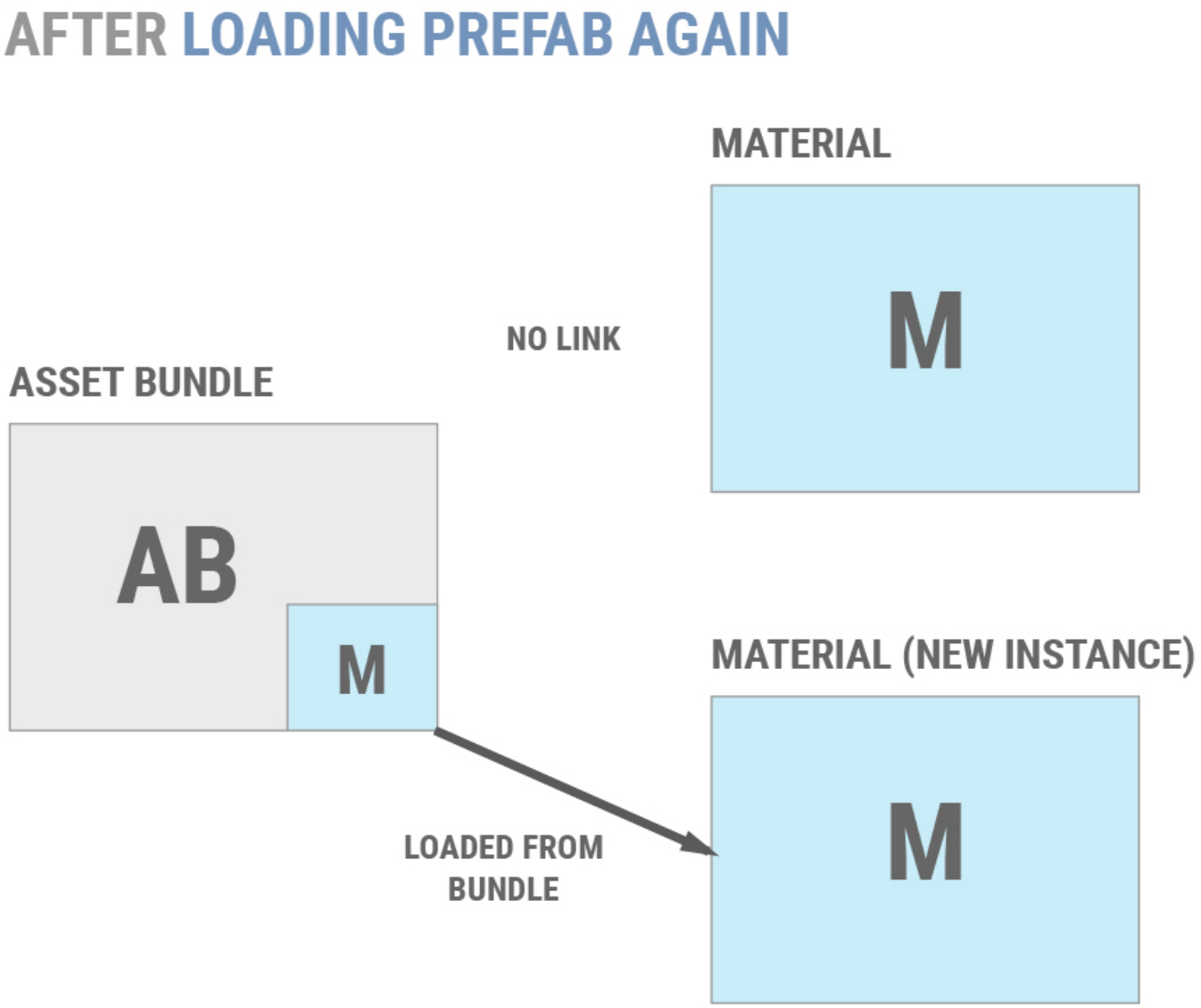
If AB.Unload(true) is called. Any instance of M in the active scene will also be unload and destroyed.

If you were instead to call AB.Unload(false) it would break the chain of the current instances of M and AB.



If AB is loaded again later and AB.LoadAsset() is called, Unity will not re-link the existing copies of M to the newly loaded Material. There will instead be two copies of M loaded.





Generally, using `AssetBundle.Unload(false)` does not lead to an ideal situation. Most projects should use `AssetBundle.Unload(true)` to keep from duplicating objects in memory.

Most projects should use `AssetBundle.Unload(true)` and adopt a method to ensure that Objects are not duplicated. Two common methods are:

Having well-defined points during the application's lifetime at which transient AssetBundles are unloaded, such as between levels or during a loading screen.

Maintaining reference-counts for individual Objects and unload AssetBundles only when all of their constituent Objects are unused. This permits an application to unload & reload individual Objects without duplicating memory.

If an application must use `AssetBundle.Unload(false)`, then individual Objects can only be unloaded in two ways:

Eliminate all references to an unwanted Object, both in the scene and in code. After this is done, call Resources.UnloadUnusedAssets.

Load a scene non-additively. This will destroy all Objects in the current scene and invoke Resources.UnloadUnusedAssets automatically.

If you'd rather not manage loading Asset Bundes, dependencies, and Assets yourself, you might find yourself in need of the AssetBundle Manager.

2017–05–15 Page published with no editorial review

AssetBundle Manager

Leave feedback

The AssetBundle Manager is deprecated and is no longer available in the **Asset Store**. You can still download it from the AssetBundleDemo Bitbucket repository.

The AssetBundle Manager is a tool made by Unity to make using AssetBundles more streamlined.

Downloading and importing the AssetBundle Manager package not only adds a new API calls for loading and using AssetBundles but it adds some Editor functionality to streamline the workflow as well. This functionality can be found under the **Assets** menu option.

This new section will contain the following options:

Simulation Mode

Enabling Simulation Mode allows the AssetBundle Manager to work with AssetBundles but does not require the bundles themselves actually be built. The editor looks to see what Assets have been assigned to AssetBundles and uses the Assets directly, instead of actually pulling them from an AssetBundle.

The main advantage of using Simulation Mode is that Assets can be modified, updated, added, and deleted without the need to re-build and deploy the AssetBundles every time.

It is worth noting that AssetBundle Variants do not work with Simulation Mode. If you need to use variants, Local AssetBundle Server is the option you need.

Local AssetBundle Server

The AssetBundle Manager can also start a Local AssetBundle Server which can be used to test AssetBundles in the editor or in local builds (including Mobile).

The stipulation to getting the Local AssetBundle Server to work is that you must create a folder called AssetBundles in the root directory of your project which is the same level as the Assets folder. Such as:

Name	Date modified	Type	Size
AssetBundles	22/07/2015 00:38	File folder	
Assets	20/07/2015 20:26	File folder	
Library	22/07/2015 00:38	File folder	
ProjectSettings	22/07/2015 00:38	File folder	
Temp	22/07/2015 00:38	File folder	

After you create the folder you'll need to build your AssetBundles to this folder. To do this, select Build AssetBundles from the new menu options. This will build them to that directory for you.

Now you have your AssetBundles built (or have decided to use Simulation Mode) and are ready to start loading AssetBundles. Let's look at the new API calls available to us through the AssetBundle Manager.

AssetBundleManager.Initialize()

This function loads the AssetBundleManifest object. You'll need to call this before you start loading in Assets using the AssetBundle Manager. In a very simplified example, initializing the AssetBundle Manager could look like this:

```
IEnumerator Start()

{
    yield return StartCoroutine(Initialize());
}
IEnumerator Initialize()
{
    var request = AssetBundleManager.Initialize();
if (request != null)
    yield return StartCoroutine(request);
}
```

The AssetBundle Manager uses this manifest you load during the Initialize() to help with a number of features behind the **scenes**, including dependency management.

Loading Assets

Let's get right down to it. You're using the AssetBundle Manager, you've initialized it, and now you're ready to load some Assets. Let's take a look at how to load the AssetBundle and instantiate an object from that bundle:

```
IEnumerator InstantiateGameObjectAsync (string assetBundleName, string assetName

{
    // Load asset from assetBundle.
    AssetBundleLoadAssetOperation request = AssetBundleManager.LoadAssetAsync(as
    if (request == null)
        yield break;
    yield return StartCoroutine(request);
    // Get the asset.
    GameObject prefab = request.GetAsset<GameObject> ();
    if (prefab != null)
        GameObject.Instantiate(prefab);
}
```

The AssetBundle Manager performs all of it's load operation asynchronously so it returns a load operation request that loads the bundle upon calling yield return StartCoroutine(request); From there all we need to do is call the GetAsset<T>() to load a game object from the AssetBundle.

Loading Scenes

If you've got an AssetBundle name assigned to a scene and you need to load that scene in you'll need to follow a slightly different code path. The pattern is relatively the same, but with slight differences. Here's how to load a scene from an AssetBundle:

```
IEnumerator InitializeLevelAsync (string levelName, bool isAdditive)

{
    // Load level from assetBundle.
    AssetBundleLoadOperation request = AssetBundleManager.LoadLevelAsync(sceneAs
    if (request == null)
        yield break;
    yield return StartCoroutine(request);
}
```

As you can see, loading scenes is also an asynchronous and LoadLevelAsync returns a load operation request which will need to be passed to a StartCoroutine in order to load the scene.

Load Variants

Loading variants using the AssetBundle Manager doesn't actually change the code need to load in the scenes or assets. All that needs to be done is set the ActiveVariants property of the AssetBundleManager.

The ActiveVariants property is an array of strings. Simply build an array of strings containing the names of the variants you created during assigning them to the Assets. Here's how to load a scene AssetBundle with the hd variant.

```
IEnumerator InitializeLevelAsync (string levelName, bool isAdditive, string[] va

{
    //Set the activeVariants.
    AssetBundleManager.ActiveVariants = variants;
    // Load level from assetBundle.
    AssetBundleLoadOperation request = AssetBundleManager.LoadLevelAsync(variant
    if (request == null)
        yield break;
    yield return StartCoroutine(request);
}
```

Where you'd pass in the string array you built up elsewhere in your code (perhaps from a button click or some other circumstances). This will load the bundles that match the set active variants if it is available.

2017–05–15 Page published with no editorial review

Patching with AssetBundles

Leave feedback

Patching AssetBundles is as simple as downloading a new AssetBundle and replacing the existing one. If `WWW.LoadFromCacheOrDownload` or `UnityWebRequest` are used to manage an application's cached AssetBundles, passing a different version parameter to the chosen API will trigger a download of the new AssetBundles.

The more difficult problem to solve in the patching system is detecting which AssetBundles to replace. A patching system requires two lists of information:

A list of the currently downloaded AssetBundles, and their versioning information
A list of the AssetBundles on the server, and their versioning information

The patcher should download the list of server-side AssetBundles and compare the AssetBundle lists. Missing AssetBundles, or AssetBundles whose versioning information has changed, should be re-downloaded.

It is also possible to write a custom system for detecting changes to AssetBundles. Most developers who write their own system choose to use an industry-standard data format for their AssetBundle file lists, such as JSON, and a standard C# class for computing checksums, such as MD5.

Unity builds AssetBundles with data ordered in a deterministic manner. This allows applications with custom downloaders to implement differential patching.

Unity does not provide any built-in mechanism for differential patching and neither `WWW.LoadFromCacheOrDownload` nor `UnityWebRequest` perform differential patching when using the built-in caching system. If differential patching is a requirement, then a custom downloader must be written.

2017–05–15 Page published with no editorial review

Troubleshooting

Leave feedback

This section describes several problems that commonly appear in projects using AssetBundles.

Asset Duplication

Unity 5's AssetBundle system will discover all dependencies of an Object when the Object is built into an AssetBundle. This is done using the **Asset** Database. This dependency information is used to determine the set of Objects that will be included in an AssetBundle.

Objects that are explicitly assigned to an AssetBundle will only be built into that AssetBundle. An Object is "explicitly assigned" when that Object's AssetImporter has its assetBundleName property set to a non-empty string.

Any Object that is not explicitly assigned in an AssetBundle will be included in all AssetBundles that contain 1 or more Objects that reference the untagged Object.

If two different Objects are assigned to two different AssetBundles, but both have references to a common dependency Object, then that dependency Object will be copied into both AssetBundles. The duplicated dependency will also be instanced, meaning that the two copies of the dependency Object will be considered different Objects with a different identifiers. This will increase the total size of the application's AssetBundles. This will also cause two different copies of the Object to be loaded into memory if the application loads both of its parents.

There are several ways to address this problem:

Ensure that Objects built into different AssetBundles do not share dependencies. Any Objects which do share dependencies can be placed into the same AssetBundle without duplicating their dependencies.

This method usually is not viable for projects with many shared dependencies. It can produce monolithic AssetBundles that must be rebuilt and re-downloaded too frequently to be convenient or efficient.

Segment AssetBundles so that no two AssetBundles that share a dependency will be loaded at the same time.

This method may work for certain types of projects, such as level-based games. However, it still unnecessarily increases the size of the project's AssetBundles, and increases both build times and loading times.

Ensure that all dependency assets are built into their own AssetBundles. This entirely eliminates the risk of duplicated assets, but also introduces complexity. The application must track dependencies between AssetBundles, and ensure that the right AssetBundles are loaded before calling any AssetBundle.LoadAsset APIs.

In Unity 5, Object dependencies are tracked via the AssetDatabase API, located in the UnityEditor namespace. As the namespace implies, this API is only available in the Unity Editor and not at runtime. AssetDatabase.GetDependencies can be used to locate all of the immediate dependencies of a specific Object or Asset. Note that these dependencies may have their own dependencies. Additionally, the AssetImporter API can be used to query the AssetBundle to which any specific Object is assigned.

By combining the AssetDatabase and AssetImporter APIs, it is possible to write an Editor script that ensures that all of an AssetBundle's direct or indirect dependencies are assigned to AssetBundles, or that no two AssetBundles share dependencies that have not been assigned to an AssetBundle. Due to the memory cost of duplicating assets, it is recommended that all projects have such a script.

Sprite Atlas Duplication

The following sections describe a quirk of Unity 5's asset dependency calculation code when used in conjunction with automatically-generated **sprite** atlases.

Any automatically-generated sprite atlas will be assigned to the AssetBundle containing the Sprite Objects from which the sprite atlas was generated. If the sprite Objects are assigned to multiple AssetBundles, then the sprite atlas will not be assigned to an AssetBundle and will be duplicated. If the Sprite Objects are not assigned to an AssetBundle, then the sprite atlas will also not be assigned to an AssetBundle.

To ensure that sprite atlases are not duplicated, check that all sprites tagged into the same sprite atlas are assigned to the same AssetBundle.

Unity 5.2.2p3 and older

Automatically-generated sprite atlases will never be assigned to an AssetBundle. Because of this, they will be included in any AssetBundles containing their constituent sprites and also any AssetBundles referencing their constituent sprites.

Because of this problem, it is strongly recommended that all Unity 5 projects using Unity's **sprite packer** upgrade to Unity 5.2.2p4, 5.3 or any newer version of Unity.

For projects that cannot upgrade, there are two workarounds for this problem:

Easy: Avoid using Unity's built-in sprite packer. Sprite atlases generated by external tools will be normal Assets, and can be properly assigned to an AssetBundle.

Hard: Assign all Objects that use automatically atlased sprites to the same AssetBundle as the sprites.

This will ensure that the generated sprite atlas is not seen as the indirect dependency of any other AssetBundles and will not be duplicated.

This solution preserves the workflow of using Unity's sprite packer, but it degrades developers' ability to separate Assets into different AssetBundles, and forces the re-download of an entire sprite atlas when any data changes on any **component** referencing the atlas, even if the atlas itself is unchanged.

Android Textures

Due to heavy device fragmentation in the Android ecosystem, it is often necessary to compress textures into several different formats. While all Android devices support ETC1, ETC1 does not support textures with alpha channels. Should an application not require OpenGL ES 2 support, the cleanest way to solve the problem is to use ETC2, which is supported by all Android OpenGL ES 3 devices.

Most applications need to ship on older devices where ETC2 support is unavailable. One way to solve this problem is with Unity 5's AssetBundle Variants. (Please see Unity's Android optimization guide for details on other options.)

To use AssetBundle Variants, all textures that cannot be cleanly compressed using ETC1 must be isolated into texture-only AssetBundles. Next, create sufficient variants of these AssetBundles to support the non-ETC2-capable slices of the Android ecosystem, using vendor-specific texture **compression** formats such as DXT5, **PVRTC** and ATITC. For each AssetBundle Variant, change the included textures' TextureImporter settings to the **compression** format appropriate to the Variant.

At runtime, support for the different **texture compression** formats can be detected using the SystemInfo.SupportsTextureFormat API. This information should be used to select and load the AssetBundle Variant containing textures compressed in a supported format.

More information on Android texture compression formats can be found here.

iOS File Handle Overuse

The issue described in the following section was fixed in Unity 5.3.2p2. Current versions of Unity are not affected by this issue.

In versions prior to Unity 5.3.2p2, Unity would hold an open file handle to an AssetBundle the entire time that the AssetBundle is loaded. This is not a problem on most platforms. However, **iOS** limits the number of file handles a process may simultaneously have open to 255. If loading an AssetBundle causes this limit to be exceeded, the loading call will fail with a “Too Many Open File Handles” error.

This was a common problem for projects trying to divide their content across many hundreds or thousands of AssetBundles.

For projects unable to upgrade to a patched version of Unity, temporary solutions are:

Reducing the number of AssetBundles in use by merging related AssetBundles
Using AssetBundle.Unload(false) to close an AssetBundle’s file handle, and managing the loaded Objects’ lifecycles manually

---

2017–05–15 Page published with no editorial review

Unity Asset Bundle Browser tool

Leave feedback

NOTE: This tool is extra functionality to Unity's standard functionality. To access it, you have to download it from GitHub and install it separately from the standard Unity Editor's download and install.

This tool enables the user to view and edit the configuration of **asset** bundles for their Unity project. It will block editing that would create invalid bundles, and inform you of any issues with existing bundles. It also provides basic build functionality.

Use this tool as an alternative to selecting assets and setting their asset bundle manually in the inspector. It can be dropped into any Unity project with a version of 5.6 or greater. It will create a new menu item in **Window** > **AssetBundle Browser**. The bundle configuration and build functionality are split into two tabs within the new window.

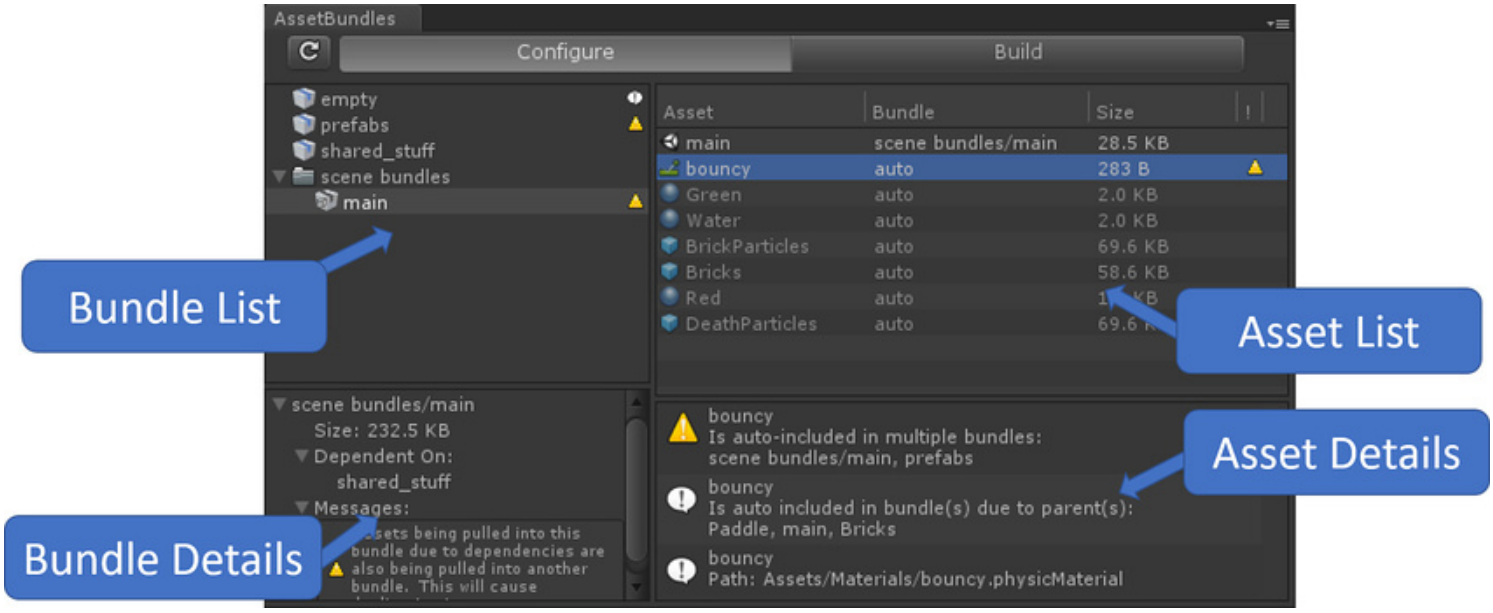
Requires Unity 5.6+

Usage - Configure

Note: this utility is in a pre-release state, and accordingly we recommend creating a backup of your project before using it.

This window provides an explorer like interface to manage and modify asset bundles in your project. When first opened, the tool will parse all bundle data in the background, slowly marking warnings or errors it detects. It does what it can to stay in sync with the project, but cannot always be aware of activity outside the tool. To force a quick pass at error detection, or to update the tool with changes made externally, hit the Refresh button in the upper left.

The window is broken into four sections: Bundle List, Bundle Details, Asset List, and Asset Details.



Bundle List

Left hand pane showing a list of all bundles in the project. Available functionality:

Select a bundle or set of bundles to see a list of the assets that will be in the bundle in the Asset List pane.

Bundles with variants are a darker grey and can be expanded to show the list of variants.

Right-click or slow-double-click to rename bundle or bundle folder.

If a bundle has any error, warning, or info message, an icon will appear on the right side. Mouse over the icon for more information.

If a bundle has at least one **scene** in it (making it a scene bundle) and non-scene assets explicitly included, it will be marked as having an error. This bundle will not build until fixed.

Bundles with duplicated assets will be marked with a warning (more information on duplication in Asset List section below)

Empty bundles will be marked with an info message. For a number of reasons, empty bundles are not very stable and can dissapear from this list at times.

Folders of bundles will be marked with the highest message from the contained bundles.

To fix the duplicated inclusion of assets in bundles, you can:

Right click on a single bundle to move all assets determined to be duplicates into a new bundle.

Right click on multiple bundles to either move the assets from all selected bundles that are duplicates into a new bundle, or only those that are shared within the selection.

You can also drag duplicate assets out of the Asset List pane into the Bundle List to explicitly include them in a bundle. More info on this in the Asset List feature set below.

Right click or hit DEL to delete bundles.

Drag bundles around to move them into and out of folders, or merge them.

Drag assets from the Project Explorer onto bundels to add them.

Drag assets onto empty space to create a new bundle.

Right click to create new bundles or bundle folders.

Right click to “Convert to Variant”

This will add a variant (initially called “newvariant”) to the selected bundle.

All assets currently in selected bundle will be moved into the new variant

ComingSoon: Mismatch detection between variants.

Icons indicate if the bundle is a standard or a scene bundle.

Icon for standard bundle

Icon for scene bundle

Bundle Details

Lower left hand pane showing details of the bundles(s) selected in the Bundle List pane. This pane will show the following information if it is available:

Total bundle size. This is a sum of the on-disk size of all assets.

Bundles that the current bundle depends on

Any messages (error/warning/info) associated with the current bundle.

Asset List

Upper right hand pane providing a list of assets contained in whichever bundles are selected in the Bundle List. Available functionality:

View all assets anticipated to be included in bundle. Sort asset list by any column header.

View assets explicitly included in bundle. These are assets that have been assigned a bundle explicitly. The **inspector** will reflect the bundle inclusion, and in this view they will say the bundle name next to the asset name.

View assets implicitly included in bundle. These assets will say *auto* as the name of the bundle next to the asset name. If looking at these assets in the inspector they will say *None* as the assigned bundle.

These assets have been added to the selected bundle(s) due to a dependency on another included asset. Only assets that are not explicitly assigned to a bundle will be implicitly included in any bundles.

Note that this list of implicit includes can be incomplete. There are known issues with materials and textures not always showing up correctly.

As multiple assets can share dependencies, it is common for a given asset to be implicitly included in multiple bundles. If the tool detects this case, it will mark both the bundle and the asset in question with a warning icon.

To fix the duplicate-inclusion warnings, you can manually move assets into a new bundle or right click the bundle and selecting one of the “Move duplicate” options.

Drag assets from the Project Explorer into this view to add them to the selected bundle. This is only valid if only one bundle is selected, and the asset type is compatable (scenes onto scene bundles, etc.)

Drag assets (explicit or implicit) from the Asset List into the Bundle List (to add them to different bundles, or a newly created bundle).

Right click or hit DEL to remove assets from bundles (does not remove assets from project).

Select or double-click assets to reveal them in the Project Explorer.

A note on including folders in bundles. It is possible to assign an asset folder (from the Project Explorer) to a bundle. When viewing this in the browser, the folder itself will be listed as explicit and the contents implicit. This reflects the priority system used to assign assets to bundles. For example, say your game had five **prefabs** in

Assets/**Prefabs**, and you marked the folder "**Prefabs**" as being in one bundle, and one of the actual **prefabs** ("PrefabA") as being in another. Once built, "PrefabA" would be in one bundle, and the other four prefabs would be in the other.

Asset Details

Lower right hand pane showing details of the asset(s) selected in the Asset List pane. This pane cannot be interacted with, but will show the following information if it is available:

Full path of asset

Reason for implicit inclusion in bundles if it is implicit.

Reason for warning if any.

Reason for error if any.

Troubleshooting

Can't rename or delet a specific bundle. This is occasionally caused when first adding this tool to an existing project. Please force a reimport of your assets through the Unity menu system to refresh the data.

External Tool Integration

Other tools that generate asset bundle data can choose to integrate with the browser. Currently the primary example is the Asset Bundle Graph Tool. If integrations are detected, then a selection bar will appear near the top of the browser. It will allow you to select the Default data source (Unity's AssetDatabase) or an integrated tool. If none are detected, the selector is not present, though you can add it by right-clicking on the tab header and selecting "Custom Sources".

Usage - Build

The Build tab provides basic build functionality to get you started using asset bundles. In most profressional scenarios, users will end up needing a more advanced build setup. All are welcome to use the build code in this tool as a starting point for writing their own once this no longer meets their needs. Interface:

Build Target - Platform the bundles will be built for

Output Path - Path for saving built bundles. By default this is AssetBundles/. You can edit the path manually, or by selecting "Browse". To return to the default naming convention, hit "Reset".

Clear Folders - This will delete all data from the build path folder prior to building.

Copy to StreamingAssets - After the build is complete, this will copy the results to Assets/StreamingAssets. This can be useful for testing, but would not be used in production.

Advanced Settings

Compression - Choose between no **compression**, standard LZMA, or chunk-based LZ4 **compression**.

Exclude Type Information - Do not include type information within the asset bundle

Force Rebuild - Rebuild bundles needing to be built. This is different than “Clear Folders” as this option will not delete bundles that no longer exist.

Ignore Type Tree Changes - Ignore the type tree changes when doing the incremental build check.

Append Hash - Append the hash to the asset bundle name.

Strict Mode - Do not allow the build to succeed if any errors are reporting during it.

Dry Run Build - Do a dry run build.

Build - Executes build.

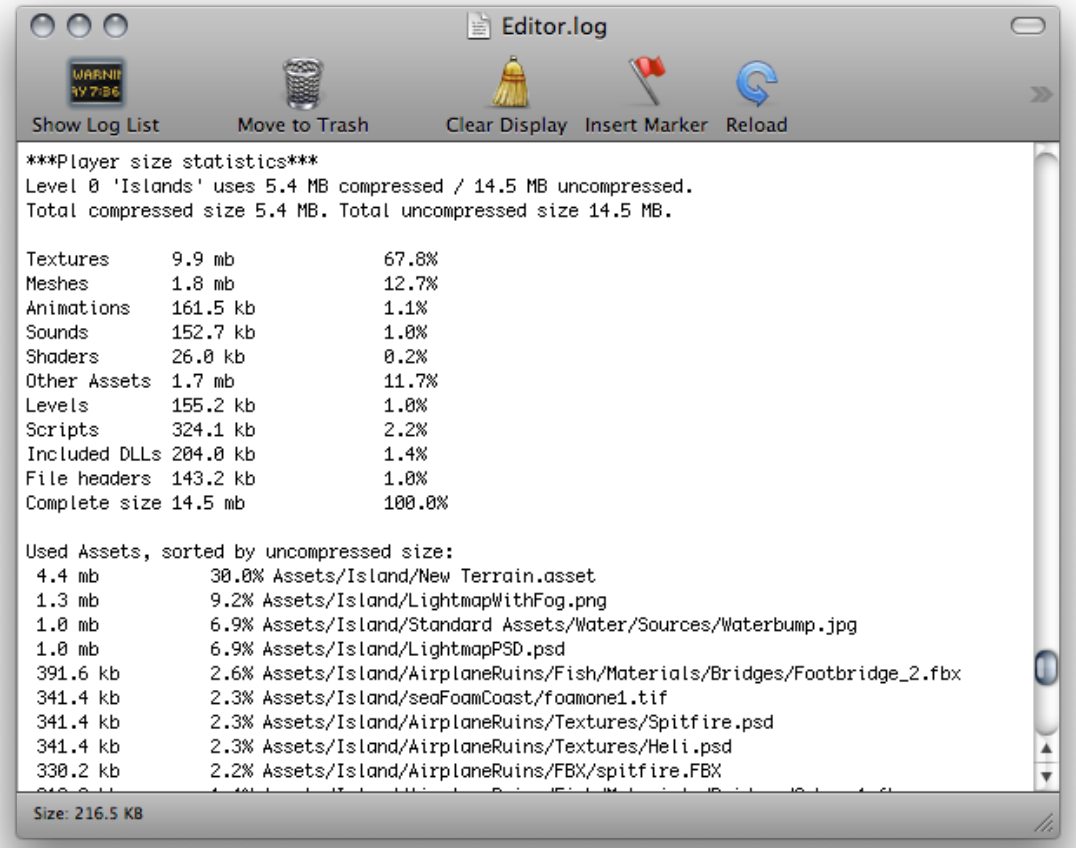
2017–05–18 Page published with limited editorial review

New feature in 5.6

Reducing the file size of your build

Leave feedback

Keeping the file size of the built app to a minimum is important, especially for mobile devices or for app stores that impose a size limit. The first step in reducing the size is to determine which Assets contribute most to it, because these Assets are the most likely candidates for optimization. This information is available in the Editor Log just after you have performed the build. Go to the Console window (menu: **Window** > **General** > **Console**), click the small drop-down panel in the top right, and select **Open Editor Log**.



The Editor Log just after a build

The Editor Log provides a summary of Assets broken down by type, and then lists all the individual Assets in order of size contribution. Typically, things like Textures, Sounds and Animations take up the most storage, while **Scripts**, Levels and **Shaders** usually have the smallest impact. The **File headers** mentioned in the list are not Assets - they are actually the extra data that is added to "raw" **Asset** files to store references and settings. The headers normally make very little difference to **Asset** size, but the value might be large if you have numerous large Assets in the Resources folder.

The Editor Log helps you identify Assets that you might want to remove or optimize, but you should consider the following before you start:

Unity re-codes imported Assets into its own internal formats, so the choice of source Asset type is not relevant. For example, if you have a multi-layer Photoshop Texture in the Project, it is flattened and compressed before building. Exporting the Texture as a .png file does not make any difference to build size, so you should stick to the format that is most convenient for you during development.

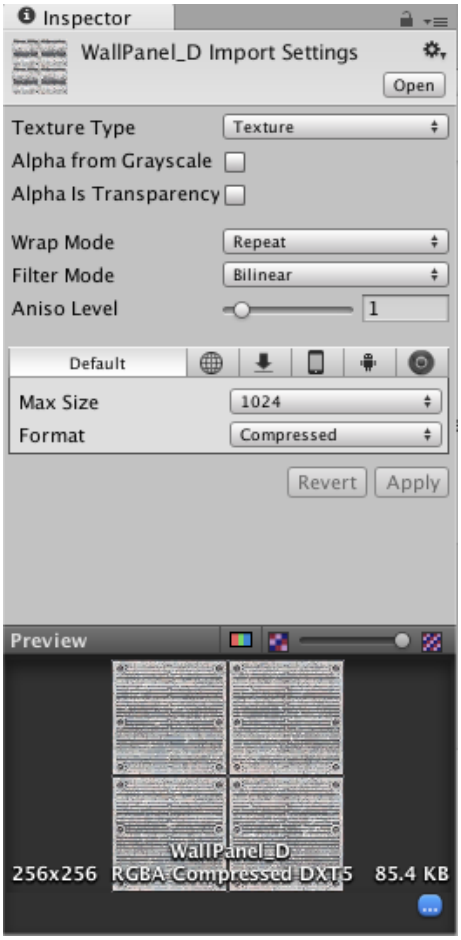
Unity strips most unused Assets during the build, so you don't gain anything by manually removing Assets from the Project. The only Assets that are not removed are scripts (which are generally very small anyway) and Assets in the Resources folder (because Unity can't determine which of these are needed and which are not). With this in mind, you should make sure that the only Assets in the Resources folder are the ones you need for the game. You might be able to replace Assets in the Resources folder with AssetBundles - this means that Unity loads Assets dynamically, thereby reducing the player size.

Suggestions for reducing build size

Textures

Textures usually take up the most space in the build. The first solution to this is to use compressed **Texture formats**. See documentation on platform-specific Texture compression for more information.

If that doesn't reduce the file size enough, try to reduce the physical size (in pixels) of the Texture images. To do this without modifying the actual source content, select the Texture in the **Project view**, and in the **Inspector** window reduce the **Max Size**. To see how this looks in-game, zoom in on a **GameObject** that uses the Texture, then adjust the **Max Size** until it starts looking worse in the **Scene view**. Changing the maximum Texture size does not affect your Texture Asset, just its resolution in the game.



Changing the Maximum Texture Size does not affect your Texture Asset, just its resolution in the game

By default, Unity compresses all Textures when importing. For faster workflow in the Editor, go to **Unity** < **Preferences** and untick the checkbox for **Compress Assets on Import**. All Textures are compressed in the build, regardless of this setting.

Meshes and Animations

You can compress Meshes and imported Animation Clips so that they take up less space in your game file. To enable **Mesh compression**, select the **Mesh**, then in the Inspector window set the **Mesh Compression** to **Low**, **Medium** or **High**. **Mesh** and Animation **compression** uses quantization, which means it takes less space, but the **compression** can introduce some inaccuracies. Experiment with what level of **compression** is acceptable for your models.

Note that Mesh compression only produces smaller data files, and does not use less memory at run time. Animation **keyframe reduction** produces smaller data files and uses less memory at run time; generally you should always have it enabled. See documentation on Animation Clips for more information about this.

DLLs

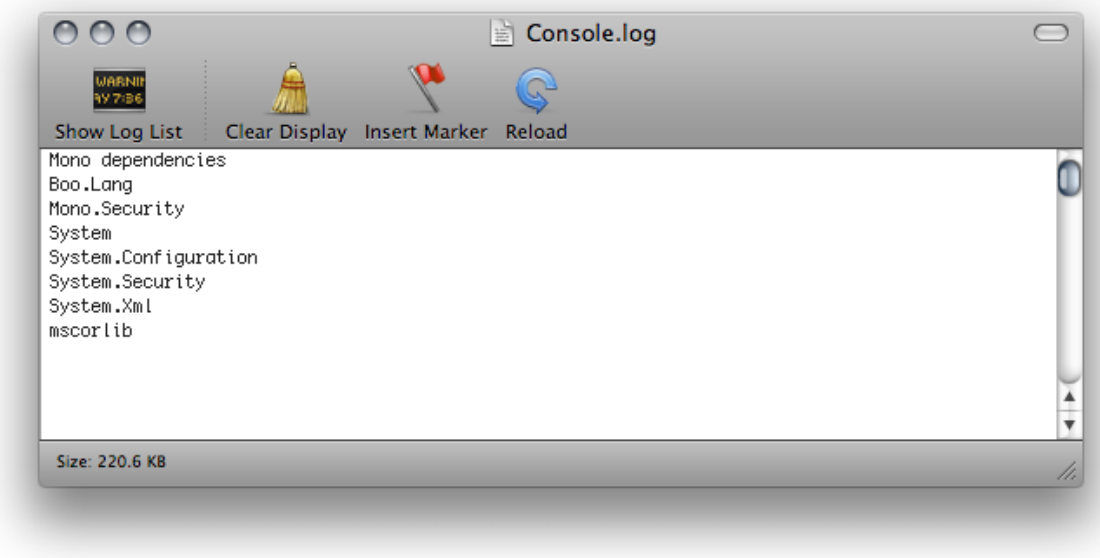
By default, Unity includes only the following DLLs in the built player:

mscorlib.dll

Boo.Lang.dll

UnityEngine.dll

When building a player, you should avoid any dependencies on *System.dll* or *System.Xml.dll*. Unity does not include these in the built player by default, but if you use their classes, they are included. These DLLs add about a megabyte to the player's storage size. If you need to parse XML in your game, you can use a library like Mono.Xml.zip as a smaller alternative to the system libraries. While most Generic containers are contained in mscorlib, Stack and few others are in System.dll, so you should avoid these if possible.



Unity includes System.Xml.dll and System.dll when building a player

Reducing mobile .NET library size

Unity supports two .NET API compatibility levels for some mobile devices: .NET 2.0 and a subset of .NET 2.0. Select the appropriate level for your build in the Player Settings.

The .NET 2.0 API profile is similar to the full .NET 2.0 API. Most library routines are fully implemented, so this option offers the best compatibility with pre-existing .NET code. However, for many games, the full library is not needed and the superfluous code takes up valuable memory space.

To avoid wasted memory, Unity also supports the .NET 2.0 Subset API profile. This is very similar to the Mono "monotouch" profile, so many limitations of the "monotouch" profile also apply to Unity's .NET 2.0 Subset profile. See the Mono Project's documentation on MonoTouch limitations for more information. Many library routines that are not commonly needed in games are left out of this profile in order to save memory. However, this also means that code with dependencies on those routines do not work correctly. This option can be a useful optimization, but you should check that existing code still works after it is applied.

Social API

Leave feedback

Social API is Unity's point of access to social features, such as:

User profiles
Friends lists
Achievements
Statistics / Leaderboards

It provides a unified interface to different social back-ends, such as **GameCenter**, and is meant to be used primarily by programmers on the game project.

The Social API is mainly an asynchronous API, and the typical way to use it is by making a function call and registering for a callback to when that function completes. The asynchronous function may have side effects, such as populating certain state variables in the API, and the callback could contain data from the server to be processed.

The Social class resides in the UnityEngine namespace and so is always available but the other Social API classes are kept in their own namespace, UnityEngine.SocialPlatforms. Furthermore, implementations of the Social API are in a sub-namespace, like SocialPlatforms.GameCenter.

Here is an example (JavaScript) of how one might use the Social API:

```
import UnityEngine.SocialPlatforms;

function Start () {
    // Authenticate and register a ProcessAuthentication callback
    // This call needs to be made before we can proceed to other calls in the So
    Social.localUser.Authenticate (ProcessAuthentication);
}

// This function gets called when Authenticate completes
// Note that if the operation is successful, Social.localUser will contain data
function ProcessAuthentication (success: boolean) {
    if (success) {
        Debug.Log ("Authenticated, checking achievements");

        // Request loaded achievements, and register a callback for processing t
        Social.LoadAchievements (ProcessLoadedAchievements);
    }
    else
        Debug.Log ("Failed to authenticate");
}

// This function gets called when the LoadAchievement call completes
function ProcessLoadedAchievements (achievements: IAchievement[]) {
    if (achievements.Length == 0)
        Debug.Log ("Error: no achievements found");
```

```
    else
        Debug.Log ("Got " + achievements.Length + " achievements");

    // You can also call into the functions like this
    Social.ReportProgress ("Achievement01", 100.0, function(result) {
        if (result)
            Debug.Log ("Successfully reported achievement progress");
        else
            Debug.Log ("Failed to report achievement");
    });
}
```

Here is the same example using C#.

```
using UnityEngine;
using UnityEngine.SocialPlatforms;

public class SocialExample : MonoBehaviour {

    void Start () {
        // Authenticate and register a ProcessAuthentication callback
        // This call needs to be made before we can proceed to other calls in th
        Social.localUser.Authenticate (ProcessAuthentication);
    }

    // This function gets called when Authenticate completes
    // Note that if the operation is successful, Social.localUser will contain d
    void ProcessAuthentication (bool success) {
        if (success) {
            Debug.Log ("Authenticated, checking achievements");

            // Request loaded achievements, and register a callback for processi
            Social.LoadAchievements (ProcessLoadedAchievements);
        }
        else
            Debug.Log ("Failed to authenticate");
    }

    // This function gets called when the LoadAchievement call completes
    void ProcessLoadedAchievements (IAchievement[] achievements) {
        if (achievements.Length == 0)
```

```
            Debug.Log ("Error: no achievements found");
        else
            Debug.Log ("Got " + achievements.Length + " achievements");

        // You can also call into the functions like this
        Social.ReportProgress ("Achievement01", 100.0, result => {
            if (result)
                Debug.Log ("Successfully reported achievement progress");
            else
                Debug.Log ("Failed to report achievement");
        });
    }
}
```

For more info on the Social API, check out the Social API Scripting Reference

JSON Serialization

Leave feedback

The JSON Serialization feature converts objects to and from JSON format. This can be useful when interacting with web services, or just for packing and unpacking data to a text-based format easily.

For information on the JsonUtility class, please see the Unity ScriptRef JsonUtility page.

Simple usage

The JSON Serialization feature is built around a notion of 'structured' JSON, which means that you describe what variables are going to be stored in your JSON data by creating a class or structure. For example:

```
[Serializable]
public class MyClass
{
    public int level;
    public float timeElapsed;
    public string playerName;
}
```

This defines a plain C# class containing three variables - **level**, **timeElapsed**, and **playerName** - and marks it as Serializable, which is necessary for it to work with the JSON serializer. You could then create an instance of it like this:

```
MyClass myObject = new MyClass();
myObject.level = 1;
myObject.timeElapsed = 47.5f;
myObject.playerName = "Dr Charles Francis";
```

And serialize it to JSON format by using `JsonUtility.ToJson`:

```
string json = JsonUtility.ToJson(myObject);
```

This would result in the `json` variable containing the string:

```
{"level":1,"timeElapsed":47.5,"playerName":"Dr Charles Francis"}
```

To convert the JSON back into an object, use `JsonUtility.FromJson`:

```
myObject = JsonUtility.FromJson<MyClass>(json);
```

This will create a new instance of `MyClass` and set the values on it using the JSON data. If the JSON data contains values that do not map to fields in `MyClass` then those values will simply be ignored, and if the JSON data is missing values for fields in `MyClass`, then those fields will be left at their constructed values in the returned object.

The JSON Serializer does not currently support working with 'unstructured' JSON (i.e. navigating and editing the JSON as an arbitrary tree of key-value pairs). If you need to do this, you should look for a more fully-featured JSON library.

Overwriting objects with JSON

It is also possible to take JSON data and deserialize it 'over' an already-created object, overwriting data that is already present:

```
JsonUtility.FromJsonOverwrite(json, myObject);
```

Any fields on the object for which the JSON does not contain a value will be left unchanged. This method allows you to keep allocations to a minimum by reusing objects that you created previously, and also to 'patch' objects by deliberately overwriting them with JSON that only contains a small subset of fields.

Note that the JSON Serializer API supports `MonoBehaviour` and `ScriptableObject` subclasses as well as plain structs/classes. However, when deserializing JSON into subclasses of `MonoBehaviour` or `ScriptableObject`, you *must* use FromJsonOverwrite; FromJson is not supported and will throw an exception.

Supported Types

The API supports any `MonoBehaviour`-subclass, `ScriptableObject`-subclass, or plain class/struct with the `[Serializable]` attribute. The object you pass in is fed to the standard Unity serializer for processing, so the same rules and limitations apply as they do in the Inspector; only fields are serialized, and types like `Dictionary<>` are not supported.

Passing other types directly to the API, for example primitive types or arrays, is not currently supported. For now you will need to wrap such types in a `class` or `struct` of some sort.

In the Editor only, there is a parallel API - `EditorJsonUtility` - which allows you to serialize any `UnityEngine.Object`-derived type both to and from JSON. This will produce JSON that contains the same data as the YAML representation of the object.

Performance

Benchmark tests have shown `JsonUtility` to be significantly faster than popular .NET JSON solutions (albeit with fewer features than some of them).

GC Memory usage is at a minimum:

`ToJson()` allocates GC memory only for the returned string.
`FromJson()` allocates GC memory only for the returned object, as well as any subobjects needed (e.g. if you deserialize an object that contains an array, then GC memory will be allocated for the array).
`FromJsonOverwrite()` allocates GC memory only as necessary for written fields (for example strings and arrays). If all fields being overwritten by the JSON are value-typed, it should not allocate any GC memory.
Using the JsonUtility API from a background thread is **permitted.** As with any multithreaded code, you should be careful not to access or alter an object on one thread while it is being serialized/deserialized on another.

Controlling the output of ToJson()

`ToJson` supports pretty-printing the JSON output. It is off by default but you can turn it on by passing `true` as the second parameter.

Fields can be omitted from the output by using the `[NonSerialized]` attribute.

Using FromJson() when the type is not known ahead of time

Deserialize the JSON into a class or struct that contains ‘common’ fields, and then use the values of those fields to work out what actual type you want. Then deserialize a second time into that type.

Streaming Assets

Leave feedback

Most **assets** in Unity are combined into the project when it is built. However, it is sometimes useful to place files into the normal filesystem on the target machine to make them accessible via a pathname. An example of this is the deployment of a movie file on **iOS** devices; the original movie file must be available from a location in the filesystem to be played by the `PlayMovie` function.

Any files placed in a folder called **StreamingAssets** (case-sensitive) in a Unity project will be copied verbatim to a particular folder on the target machine. You can retrieve the folder using the Application.streamingAssetsPath property. It's always best to use `Application.streamingAssetsPath` to get the location of the **StreamingAssets** folder, as it will always point to the correct location on the platform where the application is running.

The location of this folder varies per platform. Please note that these are case-sensitive:

On a desktop computer (Mac OS or Windows) the location of the files can be obtained with the following code:

```
path = Application.dataPath + "/StreamingAssets";
```

On iOS, use:

```
path = Application.dataPath + "/Raw";
```

On Android, use:

```
path = "jar:file://" + Application.dataPath + "!/assets/";
```

On Android, the files are contained within a compressed .jar file (which is essentially the same format as standard zip-compressed files). This means that if you do not use Unity's WWW class to retrieve the file, you need to use additional software to see inside the .jar archive and obtain the file.

Note: .dll files located in the **StreamingAssets** folder don't participate in the compilation.

ScriptableObject

Leave feedback

SWITCH TO SCRIPTING

ScriptableObject is a class that allows you to store large quantities of shared data independent from script instances. Do not confuse this class with the similarly named SerializedObject, which is an editor class and fills a different purpose. Consider for example that you have made a **prefab** with a script which has an array of a million integers. The array occupies 4MB of memory and is owned by the **prefab**. Each time you instantiate that **prefab**, you will get a copy of that array. If you created 10 game objects, then you would end up with 40MB of array data for the 10 instances.

Unity serializes all primitive types, strings, arrays, lists, types specific to Unity such as Vector3 and your custom classes with the Serializable attribute as *copies* belonging to the object they were declared in. This means that if you created a ScriptableObject and stored the million integers in an array it declares then the array will be stored with that instance. The instances are thought to own their individual data. ScriptableObject fields, or any *UnityEngine.Object* fields, such as MonoBehaviour, **Mesh**, **GameObject** and so on, are stored by *reference* as opposed to by value. If you have a script with a reference to the ScriptableObject with the million integers, Unity will only store a reference to the ScriptableObject in the script data. The ScriptableObject in turn stores the array. 10 instances of a prefab that has a reference to a ScriptableObject, that holds 4MB data, would total to roughly 4MB and not 40MB as discussed in the other example.

The intended use case for using ScriptableObject is to reduce memory usage by avoiding copies of values, but you could also use it to define pluggable data sets. An example of this would be to imagine a NPC shop in a RPG game. You could create multiple **assets** of your custom ShopContents ScriptableObject, each defining a set of items that are available for purchase. In a scenario where the game has three zones, each zone could offer different tier items. Your shop script would reference a ShopContents object that defines what items are available. Please see the scripting reference for examples.

Once you have defined a ScriptableObject-derived class, you can use the CreateAssetMenu attribute to make it easy to create custom assets using your class.

Tip: When working with ScriptableObject references in the **inspector**, you can double click the reference field to open the **inspector** for your ScriptableObject. You can also create a custom Editor to define the look of the **inspector** for your type to help manage the data that it represents.

Advanced Editor Topics

Leave feedback

This section reveals more about what goes on under the hood of the Editor, from how **Assets** and **Scenes** are stored, to customising the build pipeline and extending the Editor itself. This section will be useful to developers and teams who are comfortable with the basics of working with the Unity Editor.

Build Player Pipeline

Leave feedback

When building a player, you sometimes want to modify the built player in some way. For example you might want to add a custom icon, copy some documentation next to the player or build an Installer. You can do this via editor scripting using BuildPipeline.BuildPlayer to run the build and then follow it with whatever postprocessing code you need:-

```
// JS example.

import System.Diagnostics;

class ScriptBatch {
    @MenuItem("MyTools/Windows Build With Postprocess")
    static function BuildGame() {
        // Get filename.
        var path = EditorUtility.SaveFolderPanel("Choose Location of Built Game"
        var levels : String[] = ["Assets/Scene1.unity", "Assets/Scene2.unity"];

        // Build player.
        BuildPipeline.BuildPlayer(levels, path + "/BuiltGame.exe", BuildTarget.S

        // Copy a file from the project folder to the build folder, alongside th
        FileUtil.CopyFileOrDirectory("Assets/Templates/Readme.txt", path + "Read

        // Run the game (Process class from System.Diagnostics).
        var proc = new Process();
        proc.StartInfo.FileName = path + "/BuiltGame.exe";
        proc.Start();
    }
}
```

```
// C# example.
using UnityEditor;
using System.Diagnostics;

public class ScriptBatch
{
    [MenuItem("MyTools/Windows Build With Postprocess")]
    public static void BuildGame ()
    {
        // Get filename.
        string path = EditorUtility.SaveFolderPanel("Choose Location of Built Ga
```

```
        string[] levels = new string[] {"Assets/Scene1.unity", "Assets/Scene2.un

        // Build player.
        BuildPipeline.BuildPlayer(levels, path + "/BuiltGame.exe", BuildTarget.S

        // Copy a file from the project folder to the build folder, alongside th
        FileUtil.CopyFileOrDirectory("Assets/Templates/Readme.txt", path + "Read

        // Run the game (Process class from System.Diagnostics).
        Process proc = new Process();
        proc.StartInfo.FileName = path + "/BuiltGame.exe";
        proc.Start();
    }
}
```

PostProcessBuild Attribute

You can also use the postprocessOrder parameter of the PostProcessBuildAttribute to define the execution order for your build methods, and call your external **scripts** with the Process class from these methods as shown in the last section. This parameter is used to sort the build methods from lower to higher, and you can assign any negative or positive value to it.

Command line arguments

Leave feedback

You can run Unity from the command line (from the macOS **Terminal** or the Windows **Command Prompt**).

On **macOS**, type the following into the **Terminal** to launch Unity:

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity
```

On **Windows**, type the following into the **Command Prompt** to launch Unity:

```
C:\Program Files\Unity\Editor\Unity.exe
```

When you launch it like this, Unity receives commands and information on startup, which can be very useful for test suites, automated builds and other production tasks.
Note: Use the same method to launch standalone Unity games.

Launching Unity silently

On **macOS**, type the following into the **Terminal** to silently launch Unity:

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity -quit -batchmode -serial SB-XXX
```

Note: When activating via the command line using continuous integration (CI) tools like Jenkins, add the `-nographics` flag to prevent a WindowServer error.

On **Windows**, type the following into the **Command Prompt** to silently launch Unity:

```
"C:\Program Files\Unity\Editor\Unity.exe" -quit -batchmode -serial SB-XXXX-XXXX-XX
```

Returning the license to the license server

On **macOS**, type the following into the **Terminal** to return the license:

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity -quit -batchmode -returnlicense
```

On **Windows**, type the following into the **Command Prompt** to return the license:

```
"C:\Program Files\Unity\Editor\Unity.exe" -quit -batchmode -returnlicense
```

Create activation file and import license file by command

On **macOS**, type the following into the **Terminal**:

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity -batchmode -createManualActivat

/Applications/Unity/Unity.app/Contents/MacOS/Unity -batchmode -manualLicenseFile <
```

On **Windows**, type the following into the **Command Prompt**:

```
"C:\Program Files\Unity\Editor\Unity.exe" -batchmode -createManualActivationFile -

"C:\Program Files\Unity\Editor\Unity.exe" -batchmode -manualLicenseFile <yourulffi
```

Check Manual Activation Guide for more details.

Options

You can run the Editor and built Unity games with additional commands and information on startup. This section describes the command line options available.

Command	Details
-assetServerUpdate <IP[:port] projectName username password [r <revision>]>	Force an update of the project in the Asset Server given by `IP:port`. The port is optional, and if not given it is assumed to be the standard one (10733). It is advisable to use this command in conjunction with the `-projectPath` argument to ensure you are working with the correct project. If you don't give a project name, then the command line uses the last project opened by Unity. If no project exists at the path `-projectPath` gives, then the command line creates one automatically.

Command	Details
-batchmode	Run Unity in batch mode. You should always use this in conjunction with the other command line arguments, because it ensures no pop-up windows appear and eliminates the need for any human intervention. When an exception occurs during execution of the script code, the Asset server updates fail, or other operations fail, Unity immediately exits with return code **1**. Note that in batch mode, Unity sends a minimal version of its log output to the console. However, the Log Files still contain the full log information. You cannot open a project in batch mode while the Editor has the same project open; only a single instance of Unity can run at a time. **Tip**: To check whether you are running the Editor or Standalone Player in batch mode, use the Application.isBatchMode operator.
-buildLinux32Player <pathname>	Build a 32-bit standalone Linux player (for example, –`buildLinux32Player path/to/your/build`).
-buildLinux64Player <pathname>	Build a 64-bit standalone Linux player (for example, –`buildLinux64Player path/to/your/build`).
-buildLinuxUniversalPlayer <pathname>	Build a combined 32-bit and 64-bit standalone Linux player (for example, –`buildLinuxUniversalPlayer path/to/your/build`).
-buildOSXPlayer <pathname>	Build a 32-bit standalone Mac OSX player (for example, –`buildOSXPlayer path/to/your/build.app`).
-buildOSX64Player <pathname>	Build a 64-bit standalone Mac OSX player (for example, –`buildOSX64Player path/to/your/build.app`).
-buildOSXUniversalPlayer <pathname>	Build a combined 32-bit and 64-bit standalone Mac OSX player (for example, –`buildOSXUniversalPlayer path/to/your/build.app`).
-buildTarget <name>	Allows the selection of an active build target before loading a project. Possible options are: standalone, Win, Win64, OSXUniversal, Linux, Linux64, LinuxUniversal, iOS, Android, Web, WebStreamed, WebGL, XboxOne, PS4, WindowsStoreApps, Switch, N3DS, tvOS.
-buildWindowsPlayer <pathname>	Build a 32-bit standalone Windows player (for example, –`buildWindowsPlayer path/to/your/build.exe`).
-buildWindows64Player <pathname>	Build a 64-bit standalone Windows player (for example, –`buildWindows64Player path/to/your/build.exe`).
-stackTraceLogType	Detailed debugging feature. StackTraceLogging allows you to allow detailed logging. All settings allow **None**, **Script Only** and **Full** to be selected. (for example, –`stackTraceLogType Full`)

Command	Details
-CacheServerIPAddress <host:port>	Connect to the specified Cache Server on startup, overriding any configuration stored in the Editor Preferences. Use this to connect multiple instances of Unity to different Cache Servers.
-createProject <pathname>	Create an empty project at the given path.
-editorTestsCategories	Filter editor tests by categories. Separate test categories with a comma.
-editorTestsFilter	Filter editor tests by names. Separate test names with a comma.
-editorTestsResultFile	Path location to place the result file. If the path is a folder, the command line uses a default file name. If not specified, it places the results in the project's root folder.
-executeMethod <ClassName.MethodName>	Execute the static method as soon as Unity opens the project, and after the optional Asset server update is complete. You can use this to do tasks such as continous integration, performing Unit Tests, making builds or preparing data. To return an error from the command line process, either throw an exception which causes Unity to exit with return code **1**, or call EditorApplication.Exit with a non-zero return code. To pass parameters, add them to the command line and retrieve them inside the function using `System.Environment.GetCommandLineArgs`. To use `-executeMethod`, you need to place the enclosing script in an Editor folder. The method you execute must be defined as **static**.
-exportPackage <exportAssetPath1 exportAssetPath2 ExportAssetPath3 exportFileName>	Export a package, given a path (or set of given paths). In this example `exportAssetPath` is a folder (relative to to the Unity project root) to export from the Unity project, and `exportFileName` is the package name. Currently, this option only exports whole folders at a time. You normally need to use this command with the `-projectPath` argument.
-force-d3d11 (Windows only)	Make the Editor use Direct3D 11 for **rendering**. Normally the graphics API depends on **player settings** (typically defaults to D3D11).
-force-device-index	When using Metal, make the Editor use a particular GPU device by passing it the index of that GPU (macOS only).
-force-gfx-metal	Make the Editor use Metal as the default graphics API (macOS only).
-force-glcore	Make the Editor use OpenGL 3/4 core profile for rendering. The Editor tries to use the best OpenGL version available and all OpenGL extensions exposed by the OpenGL drivers. If the platform isn't supported, Direct3D is used.
-force-glcoreXY	Similar to `-force-glcore`, but requests a specific OpenGL context version. Accepted values for XY: 32, 33, 40, 41, 42, 43, 44 or 45.

Command	Details
-force-gles (Windows only)	Make the Editor use OpenGL for Embedded Systems for rendering. The Editor tries to use the best OpenGL ES version available, and all OpenGL ES extensions exposed by the OpenGL drivers.
-force-glesXY (Windows only)	Similar to `-force-gles`, but requests a specific OpenGL ES context version. Accepted values for XY: 30, 31 or 32.
-force-clamped	Used with `-force-glcoreXY` to prevent checking for additional OpenGL extensions, allowing it to run between platforms with the same code paths.
-force-free	Make the Editor run as if there is a free Unity license on the machine, even if a Unity Pro license is installed.
-force-low-power-device	When using Metal, make the Editor use a low power device (macOS only).
-importPackage <pathname>	Import the given package. No import dialog is shown.
-logFile <pathname>	Specify where the Editor or Windows/Linux/OSX standalone log file are written. If the path is ommitted, OSX and Linux will write output to the console. Windows uses the path *%LOCALAPPDATA%\Unity\Editor\Editor.log* as a default.
-nographics	When running in batch mode, do not initialize the graphics device at all. This makes it possible to run your automated workflows on machines that don't even have a GPU (automated workflows only work when you have a window in focus, otherwise you can't send simulated input commands). Note that `-nographics` does not allow you to bake GI, because **Enlighten** requires GPU acceleration.
-noUpm	Disables the Unity Package Manager.
-password <password>	Enter a password into the log-in form during activation of the Unity Editor.
-projectPath <pathname>	Open the project at the given path.
-quit	Quit the Unity Editor after other commands have finished executing. Note that this can cause error messages to be hidden (however, they still appear in the Editor.log file).
-returnlicense	Return the currently active license to the license server. Please allow a few seconds before the license file is removed, because Unity needs to communicate with the license server.
-runEditorTests	Run Editor tests from the project. This argument requires the `projectPath`, and it's good practice to run it with `batchmode` argument. `quit` is not required, because the Editor automatically closes down after the run is finished.

Command	Details
-serial <serial>	Activate Unity with the specified serial key. It is good practice to pass the `-batchmode` and `-quit` arguments as well, in order to quit Unity when done, if using this for automated activation of Unity. Please allow a few seconds before the license file is created, because Unity needs to communicate with the license server. Make sure that license file folder exists, and has appropriate permissions before running Unity with this argument. If activation fails, see the Editor.log for info.
-setDefaultPlatformTextureFormat	Sets the default texture compression to the desired format before importing a texture or building the project. This is so you don't have to import the texture again with the format you want. The available formats are dxt, pvrtc, atc, etc, etc2, and astc. Note that this is only supported on Android.
-silent-crashes	Prevent Unity from displaying the dialog that appears when a Standalone Player crashes. This argument is useful when you want to run the Player in automated builds or tests, where you don't want a dialog prompt to obstruct automation.
-username <username>	Enter a username into the log-in form during activation of the Unity Editor.

Command	Details

-disable-assembly-updater <assembly1 assembly2>	Specify a space-separated list of assembly names as parameters for Unity to ignore on automatic updates.
	The space-separated list of assembly names is optional: Pass the command line options without any assembly names to ignore all assemblies, as in example 1.
	
	Example 1
	`unity.exe -disable-assembly-updater`
	
	Example 2 has two assembly names, one with a pathname. Example 2 ignores `A1.dll`, no matter what folder it is stored in, and ignores `A2.dll` only if it is stored under `subfolder` folder:
	
	Example 2
	`unity.exe -disable-assembly-updater A1.dll subfolder/A2.dll`
	
	If you list an assembly in the `-disable-assembly-updater` command line parameter (or if you don't specify assemblies), Unity logs the following message to Editor.log:
	
	`[Assembly Updater] warning: Ignoring assembly [assembly_path] as requested by command line parameter.”).`
	
	Use this to avoid unnecessary API Updater overheads when importing assemblies.
	
	It is useful for importing assemblies which access a Unity API when you know the Unity API doesn't need updating. It is also useful when importing assemblies which do not access Unity APIs at all (for example, if you have built your game source code, or some of it, outside of Unity, and you want to import the resulting assemblies into your Unity project).
	
	Note: If you disable the update of any assembly that does need updating, you may get errors at compile time, run time, or both, that are hard to track.

Command	Details
-accept-apiupdate	Use this command line option to specify that APIUpdater should run when Unity is launched in batch mode. Example: `unity.exe -accept-apiupdate -batchmode [other params]` Omitting this command line argument when launching Unity in batch mode results in APIUpdater not running which can lead to compiler errors. Note that in versions prior to 2017.2 there's no way to not run APIUpdater when Unity is launched in batch mode.

Examples

C#:

```
using UnityEditor;
class MyEditorScript
{
     static void PerformBuild ()
     {
         string[] scenes = { "Assets/MyScene.unity" };
         BuildPipeline.BuildPlayer(scenes, ...);
     }
}
```

The following command executes Unity in batch mode, executes the `MyEditorScript.PerformBuild` method, and then quits upon completion.

Windows:

```
C:\program files\Unity\Editor\Unity.exe -quit -batchmode -executeMethod MyEditorSc
```

Mac OS:

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity -quit -batchmode -executeMethod
```

The following command executes Unity in batch mode, and updates from the **Asset server** using the supplied project path. The method executes after all **Assets** are downloaded and imported from the **Asset server**. After the function has finished execution, Unity automatically quits.

```
/Applications/Unity/Unity.app/Contents/MacOS/Unity -batchmode -projectPath ~/Unity
```

Unity Editor special command line arguments

You should only use these under special circumstances, or when directed by Unity Support.

Command	Details
-enableIncompatibleAssetDowngrade	Use this when you have Assets made by a newer, incompatible version of Unity, that you want to downgrade to work with your current version of Unity. When enabled, Unity presents you with a dialog asking for confirmation of this downgrade if you attempt to open a project that would require it. **Note:** This procedure is unsupported and highly risky, and should only be used as a last resort.

Unity Standalone Player command line arguments

Standalone players built with Unity also understand some command line arguments:

Command	Details
-adapter N (Windows only)	Allows the game to run full-screen on another display. The N maps to a Direct3D display adaptor. In most cases there is a one-to-one relationship between adapters and video cards. On cards that support multi-head (that is, they can drive multiple monitors from a single card) each “head” may be its own adapter.
-batchmode	Run the game in “headless” mode. The game does not display anything or accept user input. This is mostly useful for running servers for networked games.
-force-d3d11 (Windows only)	Force the game to use Direct3D 11 for rendering.
-force-d3d11-no-singlethreaded	Force DirectX 11.0 to be created without a `D3D11_CREATE_DEVICE_SINGLETHREADED` flag.
-force-device-index	Make the Standalone Player use a particular GPU device by passing it the index of that GPU (macOS only).
-force-gfx-metal	Make the Standalone Player use Metal as the default graphics API (macOS only).
-force-glcore	Force the Editor to use **OpenGL core** profile for rendering. The Editor tries to use the best OpenGL version available, and all OpenGL extensions exposed by the OpenGL drivers. If the platform isn’t supported, Direct3D is used.

Command	Details
-force-glcoreXY	Similar to `-force-glcore`, but requests a specific OpenGL context version. Accepted values for XY: 32, 33, 40, 41, 42, 43, 44 or 45.
-force-clamped	Used together with `-force-glcoreXY`, this prevents checking for additional OpenGL extensions, allowing it to run between platforms with the same code paths.
-force-low-power-device	Make the Standalone Player use a low power device (macOS only).
-force-wayland	Activate experimental Wayland support when running a Linux player.
-nographics	When running in batch mode, do not initialize graphics device at all. This makes it possible to run your automated workflows on machines that don't even have a GPU.
-nolog (Linux & Windows only)	Do not produce an output log. Normally `output_log.txt` is written in the `*_Data` folder next to the game executable, where Debug.Log output is printed.
-popupwindow	Create the window as a a pop-up window, without a frame.
-screen-fullscreen	Override the default full-screen state. This must be 0 or 1.
-screen-height	Override the default screen height. This must be an integer from a supported resolution.
-screen-width	Override the default screen width. This must be an integer from a supported resolution.
-screen-quality	Override the default screen quality. Example usage would be: `/path/to/myGame -screen-quality Beautiful`
-show-screen-selector	Forces the screen selector dialog to be shown.
-single-instance (Linux & Windows only)	Allow only one instance of the game to run at the time. If another instance is already running then launching it again with `-single-instance` focuses the existing one.
-parentHWND <HWND> delayed (Windows only)	Embed the Windows Standalone application into another application. When using this, you need to pass the parent application's window handle ('HWND') to the Windows Standalone application. When passing `-parentHWND 'HWND' delayed`, the Unity application is hidden while it is running. You must also call SetParent from the Microsoft Developer library for Unity in the application. Microsoft's `SetParent` embeds the Unity window. When it creates Unity processes, the Unity window respects the position and size provided as part of the Microsoft's STARTUPINFO structure. To resize the Unity window, check its GWLP_USERDATA in Microsoft's `GetWindowLongPtr` function. Its lowest bit is set to 1 when graphics have been initialized and it's safe to resize. Its second lowest bit is set to 1 after the Unity splash screen has finished displaying. For more information, see this example: EmbeddedWindow.zip

Universal Windows Platform command line arguments

Universal Windows Apps don't accept command line arguments by default, so to pass them you need to call a special function from **MainPage.xaml.cs/cpp** or **MainPage.cs/cpp**. For example:

```
appCallbacks.AddCommandLineArg("-nolog");
```

You should call this before the `appCallbacks.Initialize()` function.

Command	Details
-nolog	Don't produce UnityPlayer.log.
-force-driver-type-warp	Force the DirectX 11.0 driver type WARP device (see Microsoft's documentation on Windows Advanced Rasterization Platform for more information).
-force-d3d11-no-singlethreaded	Force DirectX 11.0 to be created without a `D3D11_CREATE_DEVICE_SINGLETHREADED` flag.
-force-gfx-direct	Force single threaded rendering.
-force-feature-level–9–3	Force DirectX 11.0 feature level 9.3.
-force-feature-level–10–0	Force DirectX 11.0 feature level 10.0.
-force-feature-level–10–1	Force DirectX 11.0 feature level 10.1.
-force-feature-level–11–0	Force DirectX 11.0 feature level 11.0.

2018–09–07 Page amended with limited editorial review

"accept-apiupdate" command line option added in Unity 2017.2

"-force-clamped" command line argument added in Unity 2017.3

Tizen support discontinued in 2017.3

"noUpm", "setDefaultPlatformTextureFormat" and "CacheServerIPAddress" command line options added in Unity 2018.1

"Application.isBatchMode" operator added in 2018.2

Batch mode and built-in coroutine compatibility

Leave feedback

This page describes the supported features when running the Unity Editor and Standalone Player in batch mode.

When running Unity, the following built-in coroutine operators add functionality:

AsyncOperation

WaitForEndOfFrame

WaitForFixedUpdate

WaitForSeconds

WaitForSecondsRealtime

WaitUntil

WaitWhile

The following table shows which of these operators Unity supports inside the Editor and Standalone Player, and when running each of them in batch mode using the -batchmode command line argument:

	Editor	Editor - batchmode	Unity Standalone Player	Unity Standalone Player - batchmode
`AsyncOperation`	Yes	Yes	Yes	Yes
`WaitForEndOfFrame`	Yes	No*	Yes	Yes
`WaitForFixedUpdate`	Yes	Yes	Yes	Yes
`WaitForSeconds`	Yes	Yes	Yes	Yes
`WaitForSecondsRealtime`	Yes	Yes	Yes	Yes
`WaitUntil`	Yes	Yes	Yes	Yes
`WaitWhile`	Yes	Yes	Yes	Yes

* You cannot use `WaitForEndOfFrame` when running the Editor with `-batchmode`, because systems like animation, physics and **timeline** might not work correctly in the Editor. This is because Unity does not currently update these systems when using `WaitForEndOfFrame`.

Running coroutines

Inside the Editor

In the Editor, press the “Play” button to run your code with coroutines.

Editor in batch mode

To run coroutines when launching the Editor in batch mode from the command line, enter:

C:\Program Files\Unity\Editor\Unity.exe **-projectPath** PROJECT_PATH **-batchMode**

Inside the Standalone Player

Launch your Standalone Player to run your code. The Player loads and then waits for your coroutines to complete.

Standalone Player in batch mode

To run coroutines when launching the Player in batch mode from the command line, enter:

PATH_TO_STANDALONE_BUILD **-projectPath** PROJECT_PATH **-batchMode**

For example, on Windows:

C:\projects\myproject\builds\myproject.exe **-batchMode**

On Mac:

~/UnityProjects/myproject/builds/myproject **-batchMode**

Example coroutine scripts

AsyncOperation

```
using System.Collections;
using UnityEngine;

[ExecuteInEditMode]
public class ExampleClass : MonoBehaviour
{
    public void Start()
    {
        StartCoroutine(Example_AsyncTests());
    }

    public IEnumerator Example_AsyncTests()
    {
        Debug.Log("Start of AsyncLoad Example");

        var load = UnityEngine.Resources.LoadAsync("");
        yield return load;
        yield return null;

        Debug.Log("End of AsyncLoad Example");
    }
}
```

WaitForEndOfFrame

```
using System.Collections;
using UnityEngine;

[ExecuteInEditMode]
public class ExampleClass : MonoBehaviour
{
    public void Start()
    {
        StartCoroutine(Example_WaitForEndOfFrame_Coroutine());
    }

    public IEnumerator Example_WaitForEndOfFrame_Coroutine()
    {
        Debug.Log("Start of WaitForEndOfFrame Example");

        yield return new WaitForEndOfFrame();

        Debug.Log("End of WaitForEndOfFrame Example");
    }
}
```

WaitForFixedUpdate

```
using System.Collections;
using UnityEngine;

[ExecuteInEditMode]
public class ExampleClass : MonoBehaviour
{
    public void Start()
    {
        StartCoroutine(Example_WaitForFixedUpdate_Coroutine());
    }

    public IEnumerator Example_WaitForFixedUpdate_Coroutine()
    {
        Debug.Log("Start of WaitForFixedUpdate Example");

        yield return new WaitForFixedUpdate();
```

```
        Debug.Log("End of WaitForFixedUpdate Example");
    }
}
```

WaitForSeconds

```
using System.Collections;
using UnityEngine;

[ExecuteInEditMode]
public class ExampleClass : MonoBehaviour
{
    public void Start()
    {
        StartCoroutine(Example_WaitForSeconds_Coroutine());
    }

    public IEnumerator Example_WaitForSeconds_Coroutine()
    {
        Debug.Log("Start of WaitForSeconds Example");

        yield return new WaitForSeconds(1.5f);

        Debug.Log("End of WaitForSeconds Example");
    }
}
```

WaitForSecondsRealtime

```
using System.Collections;
using UnityEngine;

[ExecuteInEditMode]
public class ExampleClass : MonoBehaviour
{
    public void Start()
    {
        StartCoroutine(Example_WaitForSecondsRealtime_Coroutine());
    }
```

```
    public IEnumerator Example_WaitForSecondsRealtime_Coroutine()
    {
        Debug.Log("Start of WaitForSecondsRealtime Example");

        yield return new WaitForSecondsRealtime(1.5f);

        Debug.Log("End of WaitForSecondsRealtime Example");
    }
}
```

WaitUntil

```
using System.Collections;
using UnityEngine;

[ExecuteInEditMode]
public class ExampleClass : MonoBehaviour
{
    public void Start()
    {
        StartCoroutine(Example_WaitUntil_Coroutine());
    }

    public IEnumerator Example_WaitUntil_Coroutine()
    {
        Debug.Log("Start of WaitUntil Example");

        yield return new WaitUntil(() => Time.time > 5.0f);

        Debug.Log("End of WaitUntil Example");
    }
}
```

WaitWhile

```
using System.Collections;
using UnityEngine;

[ExecuteInEditMode]
public class ExampleClass : MonoBehaviour
```

```
{
    public void Start()
    {
        StartCoroutine(Example_WaitWhile_Coroutine());
    }

    public IEnumerator Example_WaitWhile_Coroutine()
    {
        Debug.Log("Start of WaitWhile Example");

        yield return new WaitWhile(() => Time.time < 5.0f);

        Debug.Log("End of WaitWhile Example");
    }
}
```

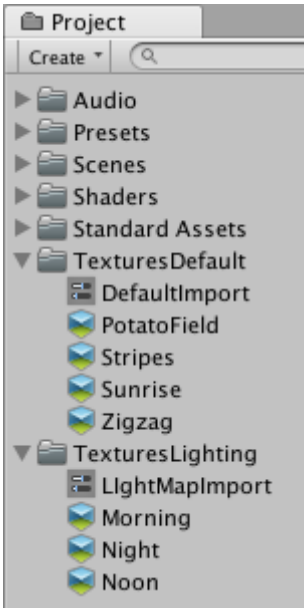
2018–06–06 Page published with editorial review

Added advice on using batchmode and coroutines in 2017.4

Applying defaults to assets by folder

Leave feedback

For large projects, you might use several Presets for importing the same type of **asset**. For example, for texture assets, you might have a Preset for importing Default textures and another for **Lightmap** textures. In the *Assets* folder of your project, you have separate folders for each of these types of textures.



The *TexturesDefault* and *TexturesLighting* folders each have a Preset

The following script applies a Preset based on the folder that you add an asset to. This script chooses the Preset that is in the same folder as the asset. If there is no Preset in the folder, this script searches parent folders. If there are no Presets in parent folders, Unity uses the default Preset that the **Preset Manager** specifies.

To use this script, create a new folder named *Editor* in the Project window, create a new C# Script in this folder, then copy and paste this script.

```
using System.IO;
using UnityEditor;
using UnityEditor.Presets;

public class PresetImportPerFolder : AssetPostprocessor
{
    void OnPreprocessAsset()
    {
        // Make sure we are applying presets the first time an asset is imported
        if (assetImporter.importSettingsMissing)
        {
            // Get the current imported asset folder.
            var path = Path.GetDirectoryName(assetPath);
            while (!string.IsNullOrEmpty(path))
            {
                // Find all Preset assets in this folder.
                var presetGuids = AssetDatabase.FindAssets("t:Preset", new[] { p
```

```
                foreach (var presetGuid in presetGuids)
                {
                    // Make sure we are not testing Presets in a subfolder.
                    string presetPath = AssetDatabase.GUIDToAssetPath(presetGuid
                    if (Path.GetDirectoryName(presetPath) == path)
                    {
                        // Load the Preset and try to apply it to the importer.
                        var preset = AssetDatabase.LoadAssetAtPath<Preset>(prese
                        if (preset.ApplyTo(assetImporter))
                            return;
                    }
                }

                // Try again in the parent folder.
                path = Path.GetDirectoryName(path);
            }
        }
    }
}
```

2017–03–27 Page published with limited editorial review New feature in 2018.1

Support for custom Menu Item and Editor features

Leave feedback

In your Editor **scripts**, use the ObjectFactory class to create new **GameObjects**, **components** and **Assets**. When creating these items, the `ObjectFactory` class automatically uses default Presets. Your script does not have to search for and apply default Presets, because `ObjectFactory` handles this for you.

Support for new types

To support and enable Presets by default, your class must inherit from one of the following:

UnityEngine.Monobehaviour
UnityEngine.ScriptableObject
UnityEngine.ScriptedImporter

The Preset **Inspector** creates a temporary instance of your class, so that users can modify its values, so make sure your class does not affect or rely on other objects such as static values, Project Assets or **Scene** instances.

Tip: Using a CustomEditor attribute is optional.

Use case example: Preset settings in a custom Editor window

When setting up a custom EditorWindow class with settings that could use Presets:

Use a ScriptableObject to store a copy of your settings. It can have a CustomEditor attribute too. The Preset system handles this object.

Always use this temporary `ScriptableObject` Inspector to show the Preset settings in your **UI**. This allows your users to have the same **UI** in your `EditorWindow` and when editing saved Presets.

Expose a Preset button and use your own PresetSelectorReceiver implementation to keep your `EditorWindow` settings up-to-date when a Preset is selected in the **Select Preset** window.

The following script examples demonstrate how to add Preset settings to a simple `EditorWindow`.

This script example demonstrates a ScriptableObject that keeps and shows settings in a custom window (saved to a file called *Editor/MyWindowSettings.cs*):

```
using UnityEngine;

// Temporary ScriptableObject used by the Preset system

public class MyWindowSettings : ScriptableObject
{
    [SerializeField]
    string m_SomeSettings;

    public void Init(MyEditorWindow window)
    {
        m_SomeSettings = window.someSettings;
    }

    public void ApplySettings(MyEditorWindow window)
    {
        window.someSettings = m_SomeSettings;
```

```
        window.Repaint();
    }
}
```

Script example of a `PresetSelectorReceiver` that updates the `ScriptableObject` used in the custom window (saved to a file called *Editor/MySettingsReceiver.cs)*:

```
using UnityEditor.Presets;

// PresetSelector receiver to update the EditorWindow with the selected values.

public class MySettingsReceiver : PresetSelectorReceiver
{
    Preset initialValues;
    MyWindowSettings currentSettings;
    MyEditorWindow currentWindow;

    public void Init(MyWindowSettings settings, MyEditorWindow window)
    {
        currentWindow = window;
        currentSettings = settings;
        initialValues = new Preset(currentSettings);
    }

    public override void OnSelectionChanged(Preset selection)
    {
        if (selection != null)
        {
            // Apply the selection to the temporary settings
            selection.ApplyTo(currentSettings);
        }
        else
        {
            // None have been selected. Apply the Initial values back to the temporary sel
            initialValues.ApplyTo(currentSettings);
        }

        // Apply the new temporary settings to our manager instance
        currentSettings.ApplySettings(currentWindow);
    }

    public override void OnSelectionClosed(Preset selection)
    {
        // Call selection change one last time to make sure you have the last selection va
        OnSelectionChanged(selection);
        // Destroy the receiver here, so you don't need to keep a reference to it.
        DestroyImmediate(this);
    }
}
```

Script example of an EditorWindow that shows custom settings using a temporary ScriptableObject Inspector and its Preset button (saved to a file called *Editor/MyEditorWindow.cs)*:

```
using UnityEngine;
using UnityEditor;
using UnityEditor.Presets;

public class MyEditorWindow : EditorWindow

{
    // get the Preset icon and a style to display it
    private static class Styles
    {
        public static GUIContent presetIcon = EditorGUIUtility.IconContent("Preset.Context
        public static GUIStyle iconButton = new GUIStyle("IconButton");

    }

    Editor m_SettingsEditor;
    MyWindowSettings m_SerializedSettings;

    public string someSettings
    {
        get { return EditorPrefs.GetString("MyEditorWindow_SomeSettings"); }
        set { EditorPrefs.SetString("MyEditorWindow_SomeSettings", value); }
    }

    // Method to open the window
    [MenuItem("Window/MyEditorWindow")]
    static void OpenWindow()
    {
        GetWindow<MyEditorWindow>();
    }

    void OnEnable()
    {
        // Create your settings now and its associated Inspector
        // that allows to create only one custom Inspector for the settings in the window
        m_SerializedSettings = ScriptableObject.CreateInstance<MyWindowSettings>();
        m_SerializedSettings.Init(this);
        m_SettingsEditor = Editor.CreateEditor(m_SerializedSettings);
    }

    void OnDisable()
    {
        Object.DestroyImmediate(m_SerializedSettings);
        Object.DestroyImmediate(m_SettingsEditor);
    }
```

```
    void OnGUI()
    {
        EditorGUILayout.BeginHorizontal();
        EditorGUILayout.LabelField("My custom settings", EditorStyles.boldLabel);
        GUILayout.FlexibleSpace();
        // create the Preset button at the end of the "MyManager Settings" line.
        var buttonPosition = EditorGUILayout.GetControlRect(false, EditorGUIUtility.single

        if (EditorGUI.DropdownButton(buttonPosition, Styles.presetIcon, FocusType.Passive,
        {
            // Create a receiver instance. This destroys itself when the window appears, s
            var presetReceiver = ScriptableObject.CreateInstance<MySettingsReceiver>();
            presetReceiver.Init(m_SerializedSettings, this);
            // Show the PresetSelector modal window. The presetReceiver updates your data.
            PresetSelector.ShowSelector(m_SerializedSettings, null, true, presetReceiver);
        }
        EditorGUILayout.EndHorizontal();

        // Draw the settings default Inspector and catch any change made to it.
        EditorGUI.BeginChangeCheck();
        m_SettingsEditor.OnInspectorGUI();

        if (EditorGUI.EndChangeCheck())
        {
            // Apply changes made in the settings editor to our instance.
            m_SerializedSettings.ApplySettings(this);
        }
    }
}
```

2017–03–27 Page published with limited editorial review New feature in 2018.1

Behind the Scenes

Leave feedback

Unity automatically imports assets and manages various kinds of additional data about them for you, such as what import settings should be used to import the asset, and where the asset is used throughout your project. Below is a description of how this process works.

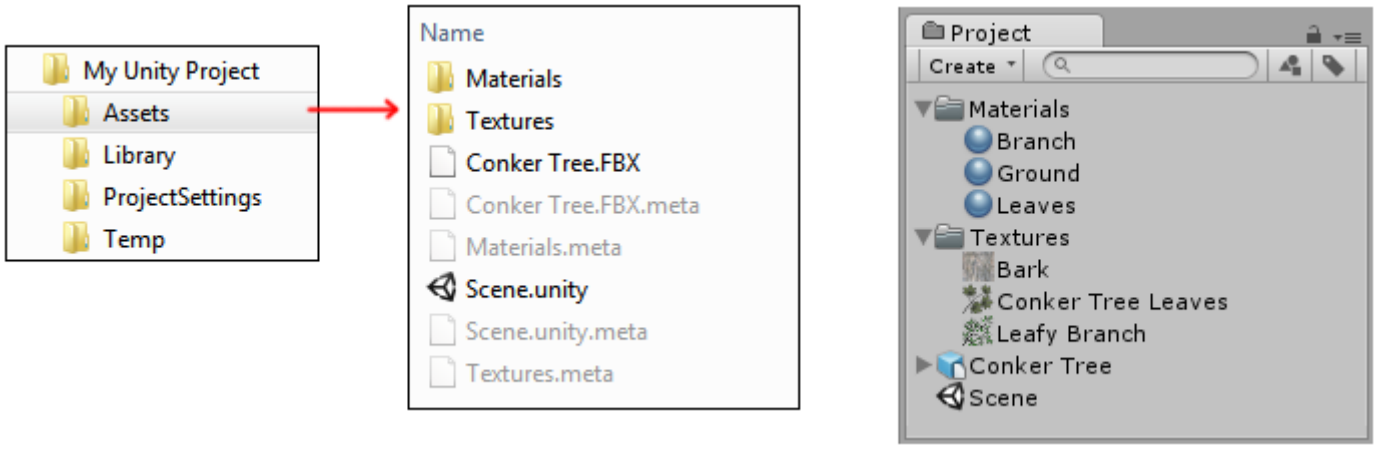
What happens when Unity imports an Asset?

1. A Unique ID is assigned

When you place an Asset such as a texture in the Assets folder, Unity will first detect that a new file has been added (the editor frequently checks the contents of the Assets folder against the list of assets it already knows about).

The first step Unity takes is to assign a unique ID to the asset. This ID is used internally by Unity to refer to the asset which means the asset can be moved or renamed without references to the asset breaking.

2. A .meta file is created



The relationship between the Assets Folder in your Unity Project on your computer, and the Project Window within Unity

You'll notice in the image above that there are **.meta** files listed in the file system for each asset and folder created within the Assets folder. These are not visible in Unity's Project Window. Unity creates these files for each asset, but they are hidden by default, so you may not see them in your Explorer/Finder either. You can make them visible by selecting this option in Unity: *Edit > Project Settings > Editor > Versions Control, Mode : **Visible Meta Files*** .

The ID that Unity assigns to each asset is stored inside the .meta file which Unity creates alongside the asset file itself. This .meta file must stay with the asset file it relates to.

IMPORTANT: *.meta files must match and stay with their respective asset files. If you move or rename an asset within Unity's own Project Window, Unity will also automatically move or rename the corresponding .meta file. If you move or rename an asset* outside* of Unity (i.e. in Windows Explorer, or Finder on the Mac), you *must* move or rename the .meta file to match.

If an asset loses its meta file (for example, if you moved or renamed the asset outside of Unity, without moving/renaming the corresponding .meta file), any reference to that asset will be broken. Unity would generate

a new .meta file for the moved/renamed asset as if it were a brand new asset, and delete the old “orphaned” .meta file.

For example, in the case of a texture asset losing its .meta file, any Materials which used that Texture will now have no reference to that texture. To fix it you would have to manually re-assign that texture to any materials which required it.

In the case of a script asset losing its .meta file, any Game Objects or **Prefabs** which had that script assigned would end up with an “unassigned script” **component**, and would lose their functionality. You would have to manually re-assign the script to these objects to fix this.

3. The source asset is processed

Unity reads and processes any files that you add to the Assets folder, converting the contents of the file to internal game-ready versions of the data. The actual asset files remain unmodified, and the processed and converted versions of the data are stored in the project’s **Library** folder.

Using internal formats for assets allows Unity to have game-ready versions of your assets ready to use at runtime in the editor, while keeping your unmodified source files in the the assets folder so that you can quickly edit them and have the changes automatically picked up by the editor. For example, the Photoshop file format is convenient to work with and can be saved directly into your Assets folder, but hardware such as mobile devices and PC graphics cards can’t accept that format directly to render as textures. All the data for Unity’s internal representation of your assets is stored in the **Library** folder which can be thought of as similar to a cache folder. As a user, you should never have to alter the Library folder manually and attempting to do so may affect the functioning of the project in the Unity editor. However, it is always safe to delete the Library folder (while the project is not open in Unity) as all its data is generated from what is stored in the Assets and ProjectSettings folders. This also means that the Library folder should not be included in **version control**.

Sometimes multiple assets are created from an import

Some asset files can result in multiple assets being created. This can occur in the following situations:

A 3D file, such as an FBX, defines Materials and/or contains embedded Textures.
In this case, the defined Materials and embedded textures are extracted and represented in Unity as separate assets.

An image file imported as multiple sprites.
It’s possible to define multiple sprites from a single graphic image, by using Unity’s Sprite Editor. In this case, each sprite defined in the editor will appear as a separate Sprite asset in the Project window.

A 3D file contains multiple animation timelines, or has multiple separate clips defined within its animation import settings.
In this situation, the multiple animation clips will appear as separate **Animation Clip** assets in the project window.

The import settings can alter the processing of the asset

As well as the unique ID assigned to the asset, the meta files contain values for all the import settings you see in the **inspector** when you have an asset selected in your project window. For a texture, this includes settings such as the Texture Type, Wrap Mode, Filter Mode and **Aniso Level**.

If you change the import settings for an asset, those changed settings are stored in the .meta file accompanying the asset. The asset will be re-imported according to your new settings, and the corresponding imported "game-ready" data will be updated in the project's Library folder.

When backing up a project, or adding a project to a Version Control Repository, you should include the main Unity project folder, containing both the **Assets** and **ProjectSettings** folders. All the information in these folders is crucial to the way Unity works. You should omit the **Library** and **Temp** folders for backup purposes.

Note: *Projects created in Unity 4.2 and earlier may not have .meta files if not explicitly enabled. Deleting the Library folder in these projects will lead to data loss and permanent project corruption because both the generated internal formats of your assets* ***and*** *the meta data were stored in the Library folder.*

AssetDatabase

Leave feedback

AssetDatabase is an API which allows you to access the assets contained in your project. Among other things, it provides methods to find and load assets and also to create, delete and modify them. The Unity Editor uses the AssetDatabase internally to keep track of **asset** files and maintain the linkage between assets and objects that reference them. Since Unity needs to keep track of all changes to the project folder, you should always use the AssetDatabase API rather than the filesystem if you want to access or modify **asset** data.

The AssetDatabase interface is only available in the editor and has no function in the built player. Like all other editor classes, it is only available to **scripts** placed in the Editor folder (just create a folder named "Editor" in the main Assets folder of your project if there isn't one already).

Importing an Asset

Unity normally imports assets automatically when they are dragged into the project but it is also possible to import them under script control. To do this you can use the AssetDatabase.ImportAsset method as in the example below.

```
using UnityEngine;
using UnityEditor;

public class ImportAsset {
    [MenuItem ("AssetDatabase/ImportExample")]
    static void ImportExample ()
    {
        AssetDatabase.ImportAsset("Assets/Textures/texture.jpg", ImportAssetOptions.Defa
    }
}
```

You can also pass an extra parameter of type AssetDatabase.ImportAssetOptions to the AssetDatabase.ImportAsset call. The scripting reference page documents the different options and their effects on the function's behaviour.

Loading an Asset

The editor loads assets only as needed, say if they are added to the **scene** or edited from the **Inspector** panel. However, you can load and access assets from a script using AssetDatabase.LoadAssetAtPath, AssetDatabase.LoadMainAssetAtPath, AssetDatabase.LoadAllAssetRepresentationsAtPath and AssetDatabase.LoadAllAssetsAtPath. See the scripting documentation for further details.

```
using UnityEngine;
using UnityEditor;

public class ImportAsset {
    [MenuItem ("AssetDatabase/LoadAssetExample")]
    static void ImportExample ()
    {
        Texture2D t = AssetDatabase.LoadAssetAtPath("Assets/Textures/texture.jpg", typec
    }
```

```
}
```

File Operations using the AssetDatabase

Since Unity keeps metadata about asset files, you should never create, move or delete them using the filesystem. Instead, you can use AssetDatabase.Contains, AssetDatabase.CreateAsset, AssetDatabase.CreateFolder, AssetDatabase.RenameAsset, AssetDatabase.CopyAsset, AssetDatabase.MoveAsset, AssetDatabase.MoveAssetToTrash and AssetDatabase.DeleteAsset.

```
public class AssetDatabaseIOExample {
    [MenuItem ("AssetDatabase/FileOperationsExample")]
    static void Example ()
    {
        string ret;

        // Create
        Material material = new Material (Shader.Find("Specular"));
        AssetDatabase.CreateAsset(material, "Assets/MyMaterial.mat");
        if(AssetDatabase.Contains(material))
            Debug.Log("Material asset created");

        // Rename
        ret = AssetDatabase.RenameAsset("Assets/MyMaterial.mat", "MyMaterialNew");
        if(ret == "")
            Debug.Log("Material asset renamed to MyMaterialNew");
        else
            Debug.Log(ret);

        // Create a Folder
        ret = AssetDatabase.CreateFolder("Assets", "NewFolder");
        if(AssetDatabase.GUIDToAssetPath(ret) != "")
            Debug.Log("Folder asset created");
        else
            Debug.Log("Couldn't find the GUID for the path");

        // Move
        ret = AssetDatabase.MoveAsset(AssetDatabase.GetAssetPath(material), "Assets/NewF
        if(ret == "")
            Debug.Log("Material asset moved to NewFolder/MyMaterialNew.mat");
        else
            Debug.Log(ret);

        // Copy
        if(AssetDatabase.CopyAsset(AssetDatabase.GetAssetPath(material), "Assets/MyMater
            Debug.Log("Material asset copied as Assets/MyMaterialNew.mat");
        else
            Debug.Log("Couldn't copy the material");
```

```
        // Manually refresh the Database to inform of a change
        AssetDatabase.Refresh();
        Material MaterialCopy = AssetDatabase.LoadAssetAtPath("Assets/MyMaterialNew.mat"

        // Move to Trash
        if(AssetDatabase.MoveAssetToTrash(AssetDatabase.GetAssetPath(MaterialCopy)))
            Debug.Log("MaterialCopy asset moved to trash");

        // Delete
        if(AssetDatabase.DeleteAsset(AssetDatabase.GetAssetPath(material)))
            Debug.Log("Material asset deleted");
        if(AssetDatabase.DeleteAsset("Assets/NewFolder"))
            Debug.Log("NewFolder deleted");

        // Refresh the AssetDatabase after all the changes
        AssetDatabase.Refresh();
    }
}
```

Using AssetDatabase.Refresh

When you have finished modifying assets, you should call AssetDatabase.Refresh to commit your changes to the database and make them visible in the project.

Text-Based Scene Files

Leave feedback

As well as the default binary format, Unity also provides a text-based format for scene data. This can be useful when working with **version control** software, since text files generated separately can be merged more easily than binary files. Also, the text data can be generated and parsed by tools, making it possible to create and analyze **scenes** automatically. The pages in this section provide some reference material for working with the format.

See the Editor Settings page for how to enable this feature.

Description of the Format

Leave feedback

Unity's **scene** format is implemented with the YAML data serialization language. While we can't cover YAML in depth here, it is an open format and its specification is available for free at the YAML website. Each object in the scene is written to the file as a separate YAML document, which is introduced in the file by the — sequence. Note that in this context, the term "object" refers to **GameObjects**, **Components** and other scene data collectively; each of these items requires its own YAML document in the scene file. The basic structure of a serialized object can be understood from an example:-

```
--- !u!1 &6
GameObject:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  importerVersion: 3
  m_Component:
  - 4: {fileID: 8}
  - 33: {fileID: 12}
  - 65: {fileID: 13}
  - 23: {fileID: 11}
  m_Layer: 0
  m_Name: Cube
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
```

The first line contains the string "!u!1 &6" after the document marker. The first number after the "!u!" part indicates the class of the object (in this case, it is a GameObject). The number following the ampersand is an object ID number which is unique within the file, although the number is assigned to each object arbitrarily. Each of the object's serializable properties is denoted by a line like the following:-

```
m_Name: Cube
```

Properties are typically prefixed with "m_" but otherwise follow the name of the property as defined in the script reference. A second object, defined further down in the file, might look something like this:-

```
--- !u!4 &8
Transform:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 6}
  m_LocalRotation: {x: 0.000000, y: 0.000000, z: 0.000000, w: 1.000000}
  m_LocalPosition: {x: -2.618721, y: 1.028581, z: 1.131627}
  m_LocalScale: {x: 1.000000, y: 1.000000, z: 1.000000}
  m_Children: []
  m_Father: {fileID: 0}
```

This is a **Transform component** attached to the GameObject defined by the YAML document above. The attachment is denoted by the line:-

```
 m_GameObject: {fileID: 6}
```

...since the GameObject's object ID within the file was 6.

Floating point numbers can be represented in a decimal representation or as a hexadecimal number in IEEE 754 format (denoted by a 0x prefix). The IEEE 754 representation is used for lossless encoding of values, and is used by Unity when writing floating point values which don't have a short decimal representation. When Unity writes numbers in hexadecimal, it will always also write the decimal format in parentheses for debugging purposes, but only the hex is actually parsed when loading the file. If you wish to edit such values manually, simply remove the hex and enter only a decimal number. Here are some valid representations of floating point values (all representing the number one):

```
myValue: 0x3F800000
myValue: 1
myValue: 1.000
myValue: 0x3f800000(1)
myValue: 0.1e1
```

An Example of a YAML Scene File

Leave feedback

An example of a simple but complete **scene** is given below. The **scene** contains just a **camera** and a cube object. Note that the file **must** start with the two lines

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
```

...in order to be accepted by Unity. Otherwise, the import process is designed to be tolerant of omissions - default values will be supplied for missing property data as far as possible.

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!header
SerializedFile:
  m_TargetPlatform: 4294967294
  m_UserInformation:
--- !u!29 &1
Scene:
  m_ObjectHideFlags: 0
  m_PVSData:
  m_QueryMode: 1
  m_PVSObjectsArray: []
  m_PVSPortalsArray: []
  m_ViewCellSize: 1.000000
--- !u!104 &2
RenderSettings:
  m_Fog: 0
  m_FogColor: {r: 0.500000, g: 0.500000, b: 0.500000, a: 1.000000}
  m_FogMode: 3
  m_FogDensity: 0.010000
  m_LinearFogStart: 0.000000
  m_LinearFogEnd: 300.000000
  m_AmbientLight: {r: 0.200000, g: 0.200000, b: 0.200000, a: 1.000000}
  m_SkyboxMaterial: {fileID: 0}
  m_HaloStrength: 0.500000
  m_FlareStrength: 1.000000
  m_HaloTexture: {fileID: 0}
  m_SpotCookie: {fileID: 0}
  m_ObjectHideFlags: 0
--- !u!127 &3
GameManager:
```

```
  m_ObjectHideFlags: 0
--- !u!157 &4
LightmapSettings:
  m_ObjectHideFlags: 0
  m_LightProbeCloud: {fileID: 0}
  m_Lightmaps: []
  m_LightmapsMode: 1
  m_BakedColorSpace: 0
  m_UseDualLightmapsInForward: 0
  m_LightmapEditorSettings:
    m_Resolution: 50.000000
    m_LastUsedResolution: 0.000000
    m_TextureWidth: 1024
    m_TextureHeight: 1024
    m_BounceBoost: 1.000000
    m_BounceIntensity: 1.000000
    m_SkyLightColor: {r: 0.860000, g: 0.930000, b: 1.000000, a: 1.000000}
    m_SkyLightIntensity: 0.000000
    m_Quality: 0
    m_Bounces: 1
    m_FinalGatherRays: 1000
    m_FinalGatherContrastThreshold: 0.050000
    m_FinalGatherGradientThreshold: 0.000000
    m_FinalGatherInterpolationPoints: 15
    m_AOAmount: 0.000000
    m_AOMaxDistance: 0.100000
    m_AOContrast: 1.000000
    m_TextureCompression: 0
    m_LockAtlas: 0
--- !u!196 &5
NavMeshSettings:
  m_ObjectHideFlags: 0
  m_BuildSettings:
    cellSize: 0.200000
    cellHeight: 0.100000
    agentSlope: 45.000000
    agentClimb: 0.900000
    ledgeDropHeight: 0.000000
    maxJumpAcrossDistance: 0.000000
    agentRadius: 0.400000
    agentHeight: 1.800000
    maxEdgeLength: 12
    maxSimplificationError: 1.300000
    regionMinSize: 8
    regionMergeSize: 20
    tileSize: 500
    detailSampleDistance: 6.000000
```

```
    detailSampleMaxError: 1.000000
    accuratePlacement: 0
  m_NavMesh: {fileID: 0}
--- !u!1 &6
GameObject:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  importerVersion: 3
  m_Component:
  - 4: {fileID: 8}
  - 33: {fileID: 12}
  - 65: {fileID: 13}
  - 23: {fileID: 11}
  m_Layer: 0
  m_Name: Cube
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!1 &7
GameObject:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  importerVersion: 3
  m_Component:
  - 4: {fileID: 9}
  - 20: {fileID: 10}
  - 92: {fileID: 15}
  - 124: {fileID: 16}
  - 81: {fileID: 14}
  m_Layer: 0
  m_Name: Main Camera
  m_TagString: MainCamera
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!4 &8
Transform:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 6}
  m_LocalRotation: {x: 0.000000, y: 0.000000, z: 0.000000, w: 1.000000}
```

```
  m_LocalPosition: {x: -2.618721, y: 1.028581, z: 1.131627}
  m_LocalScale: {x: 1.000000, y: 1.000000, z: 1.000000}
  m_Children: []
  m_Father: {fileID: 0}
--- !u!4 &9
Transform:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_LocalRotation: {x: 0.000000, y: 0.000000, z: 0.000000, w: 1.000000}
  m_LocalPosition: {x: 0.000000, y: 1.000000, z: -10.000000}
  m_LocalScale: {x: 1.000000, y: 1.000000, z: 1.000000}
  m_Children: []
  m_Father: {fileID: 0}
--- !u!20 &10
Camera:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_Enabled: 1
  importerVersion: 2
  m_ClearFlags: 1
  m_BackGroundColor: {r: 0.192157, g: 0.301961, b: 0.474510, a: 0.019608}
  m_NormalizedViewPortRect:
    importerVersion: 2
    x: 0.000000
    y: 0.000000
    width: 1.000000
    height: 1.000000
  near clip plane: 0.300000
  far clip plane: 1000.000000
  field of view: 60.000000
  orthographic: 0
  orthographic size: 100.000000
  m_Depth: -1.000000
  m_CullingMask:
    importerVersion: 2
    m_Bits: 4294967295
  m_RenderingPath: -1
  m_TargetTexture: {fileID: 0}
  m_HDR: 0
--- !u!23 &11
Renderer:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
```

```
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 6}
  m_Enabled: 1
  m_CastShadows: 1
  m_ReceiveShadows: 1
  m_LightmapIndex: 255
  m_LightmapTilingOffset: {x: 1.000000, y: 1.000000, z: 0.000000, w: 0.000000}
  m_Materials:
  - {fileID: 10302, guid: 0000000000000000e000000000000000, type: 0}
  m_SubsetIndices:
  m_StaticBatchRoot: {fileID: 0}
  m_LightProbeAnchor: {fileID: 0}
  m_UseLightProbes: 0
  m_ScaleInLightmap: 1.000000
--- !u!33 &12
MeshFilter:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 6}
  m_Mesh: {fileID: 10202, guid: 0000000000000000e000000000000000, type: 0}
--- !u!65 &13
BoxCollider:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 6}
  m_Material: {fileID: 0}
  m_IsTrigger: 0
  m_Enabled: 1
  importerVersion: 2
  m_Size: {x: 1.000000, y: 1.000000, z: 1.000000}
  m_Center: {x: 0.000000, y: 0.000000, z: 0.000000}
--- !u!81 &14
AudioListener:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_Enabled: 1
--- !u!92 &15
Behaviour:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_Enabled: 1
```

```
--- !u!124 &16
Behaviour:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_GameObject: {fileID: 7}
  m_Enabled: 1
--- !u!1026 &17
HierarchyState:
  m_ObjectHideFlags: 0
  expanded: []
  selection: []
  scrollposition_x: 0.000000
  scrollposition_y: 0.000000
```

YAML Class ID Reference

Leave feedback

A reference of common class ID numbers used by the YAML file format is given below, both in numerical order of class IDs and alphabetical order of class names. Note that some ranges of numbers are intentionally omitted from the sequence - these may represent classes that have been removed from the API or may be reserved for future use. Classes defined from **scripts** will always have class ID 114 (MonoBehaviour).

Classes Ordered by ID Number

ID	Class
1	**GameObject**
2	**Component**
3	LevelGameManager
4	Transform
5	TimeManager
6	GlobalGameManager
8	Behaviour
9	GameManager
11	AudioManager
12	ParticleAnimator
13	InputManager
15	EllipsoidParticleEmitter
17	Pipeline
18	EditorExtension
19	Physics2DSettings
20	**Camera**
21	Material
23	MeshRenderer
25	Renderer
26	ParticleRenderer
27	Texture
28	Texture2D
29	SceneSettings
30	GraphicsSettings
33	MeshFilter
41	OcclusionPortal
43	**Mesh**
45	**Skybox**
47	QualitySettings
48	**Shader**
49	TextAsset
50	Rigidbody2D
51	Physics2DManager

ID	Class
53	Collider2D
54	**Rigidbody**
55	PhysicsManager
56	**Collider**
57	**Joint**
58	CircleCollider2D
59	HingeJoint
60	PolygonCollider2D
61	BoxCollider2D
62	PhysicsMaterial2D
64	MeshCollider
65	BoxCollider
66	SpriteCollider2D
68	EdgeCollider2D
72	ComputeShader
74	AnimationClip
75	ConstantForce
76	WorldParticleCollider
78	TagManager
81	AudioListener
82	AudioSource
83	AudioClip
84	RenderTexture
87	MeshParticleEmitter
88	ParticleEmitter
89	**Cubemap**
90	**Avatar**
91	AnimatorController
92	GUILayer
93	RuntimeAnimatorController
94	ScriptMapper
95	Animator
96	TrailRenderer
98	DelayedCallManager
102	TextMesh
104	RenderSettings
108	Light
109	**CGProgram**
110	BaseAnimationTrack
111	Animation
114	MonoBehaviour

ID	Class
115	MonoScript
116	MonoManager
117	Texture3D
118	NewAnimationTrack
119	Projector
120	LineRenderer
121	Flare
122	Halo
123	LensFlare
124	FlareLayer
125	HaloLayer
126	NavMeshAreas
127	HaloManager
128	Font
129	PlayerSettings
130	NamedObject
131	GUITexture
132	GUIText
133	GUIElement
134	PhysicMaterial
135	SphereCollider
136	CapsuleCollider
137	SkinnedMeshRenderer
138	FixedJoint
140	RaycastCollider
141	BuildSettings
142	AssetBundle
143	CharacterController
144	CharacterJoint
145	SpringJoint
146	WheelCollider
147	ResourceManager
148	NetworkView
149	NetworkManager
150	PreloadData
152	MovieTexture
153	ConfigurableJoint
154	TerrainCollider
155	MasterServerInterface
156	TerrainData
157	LightmapSettings

ID	Class
158	WebCamTexture
159	EditorSettings
160	InteractiveCloth
161	ClothRenderer
162	EditorUserSettings
163	SkinnedCloth
164	AudioReverbFilter
165	AudioHighPassFilter
166	AudioChorusFilter
167	AudioReverbZone
168	AudioEchoFilter
169	AudioLowPassFilter
170	AudioDistortionFilter
171	SparseTexture
180	AudioBehaviour
181	AudioFilter
182	WindZone
183	Cloth
184	SubstanceArchive
185	ProceduralMaterial
186	ProceduralTexture
191	OffMeshLink
192	OcclusionArea
193	Tree
194	NavMeshObsolete
195	NavMeshAgent
196	NavMeshSettings
197	LightProbesLegacy
198	ParticleSystem
199	ParticleSystemRenderer
200	ShaderVariantCollection
205	LODGroup
206	BlendTree
207	Motion
208	NavMeshObstacle
210	TerrainInstance
212	SpriteRenderer
213	**Sprite**
214	CachedSpriteAtlas
215	ReflectionProbe
216	ReflectionProbes

ID	Class
218	**Terrain**
220	LightProbeGroup
221	AnimatorOverrideController
222	CanvasRenderer
223	Canvas
224	RectTransform
225	CanvasGroup
226	BillboardAsset
227	BillboardRenderer
228	SpeedTreeWindAsset
229	AnchoredJoint2D
230	Joint2D
231	SpringJoint2D
232	DistanceJoint2D
233	HingeJoint2D
234	SliderJoint2D
235	WheelJoint2D
238	NavMeshData
240	AudioMixer
241	AudioMixerController
243	AudioMixerGroupController
244	AudioMixerEffectController
245	AudioMixerSnapshotController
246	PhysicsUpdateBehaviour2D
247	ConstantForce2D
248	Effector2D
249	AreaEffector2D
250	PointEffector2D
251	PlatformEffector2D
252	SurfaceEffector2D
258	LightProbes
271	SampleClip
272	AudioMixerSnapshot
273	AudioMixerGroup
290	AssetBundleManifest
1001	**Prefab**
1002	EditorExtensionImpl
1003	AssetImporter
1004	AssetDatabase
1005	Mesh3DSImporter
1006	TextureImporter

ID	Class
1007	ShaderImporter
1008	ComputeShaderImporter
1011	AvatarMask
1020	AudioImporter
1026	HierarchyState
1027	GUIDSerializer
1028	AssetMetaData
1029	DefaultAsset
1030	DefaultImporter
1031	TextScriptImporter
1032	SceneAsset
1034	NativeFormatImporter
1035	MonoImporter
1037	AssetServerCache
1038	LibraryAssetImporter
1040	ModelImporter
1041	FBXImporter
1042	TrueTypeFontImporter
1044	MovieImporter
1045	EditorBuildSettings
1046	DDSImporter
1048	InspectorExpandedState
1049	AnnotationManager
1050	PluginImporter
1051	EditorUserBuildSettings
1052	PVRImporter
1053	ASTCImporter
1054	KTXImporter
1101	AnimatorStateTransition
1102	AnimatorState
1105	HumanTemplate
1107	AnimatorStateMachine
1108	PreviewAssetType
1109	AnimatorTransition
1110	SpeedTreeImporter
1111	AnimatorTransitionBase
1112	SubstanceImporter
1113	LightmapParameters
1120	LightmapSnapshot

Classes Ordered Alphabetically

Class	ID
ASTCImporter	1053
AnchoredJoint2D	229
Animation	111
AnimationClip	74
Animator	95
AnimatorController	91
AnimatorOverrideController	221
AnimatorState	1102
AnimatorStateMachine	1107
AnimatorStateTransition	1101
AnimatorTransitionBase	1111
AnimatorTransition	1109
AnnotationManager	1049
AreaEffector2D	249
AssetBundle	142
AssetBundleManifest	290
AssetDatabase	1004
AssetImporter	1003
AssetMetaData	1028
AssetServerCache	1037
AudioBehaviour	180
AudioChorusFilter	166
AudioClip	83
AudioDistortionFilter	170
AudioEchoFilter	168
AudioFilter	181
AudioHighPassFilter	165
AudioImporter	1020
AudioListener	81
AudioLowPassFilter	169
AudioManager	11
AudioMixer	240
AudioMixerController	241
AudioMixerEffectController	244
AudioMixerGroup	273
AudioMixerGroupController	243
AudioMixerSnapshot	272
AudioMixerSnapshotController	245
AudioReverbFilter	164
AudioReverbZone	167
AudioSource	82

Class	ID
Avatar	90
AvatarMask	1011
BaseAnimationTrack	110
Behaviour	8
BillboardAsset	226
BillboardRenderer	227
BlendTree	206
BoxCollider	65
BoxCollider2D	61
BuildSettings	141
CachedSpriteAtlas	214
Camera	20
Canvas	223
CanvasGroup	225
CanvasRenderer	222
CapsuleCollider	136
CGProgram	109
CharacterController	143
CharacterJoint	144
CircleCollider2D	58
Cloth	183
ClothRenderer	161
Collider	56
Collider2D	53
Component	2
ComputeShader	72
ComputeShaderImporter	1008
ConfigurableJoint	153
ConstantForce	75
ConstantForce2D	247
Cubemap	89
DDSImporter	1046
DefaultAsset	1029
DefaultImporter	1030
DelayedCallManager	98
DistanceJoint2D	232
EdgeCollider2D	68
EditorBuildSettings	1045
EditorExtension	18
EditorExtensionImpl	1002
EditorSettings	159

Class	ID
EditorUserBuildSettings	1051
EditorUserSettings	162
Effector2D	248
EllipsoidParticleEmitter	15
FBXImporter	1041
FixedJoint	138
Flare	121
FlareLayer	124
Font	128
GameManager	9
GameObject	1
GlobalGameManager	6
GraphicsSettings	30
GUIDSerializer	1027
GUIElement	133
GUILayer	92
GUIText	132
GUITexture	131
Halo	122
HaloLayer	125
HaloManager	127
HierarchyState	1026
HingeJoint	59
HingeJoint2D	233
HumanTemplate	1105
InputManager	13
InspectorExpandedState	1048
InteractiveCloth	160
Joint	57
Joint2D	230
KTXImporter	1054
LensFlare	123
LevelGameManager	3
LibraryAssetImporter	1038
Light	108
LightmapParameters	1113
LightmapSettings	157
LightmapSnapshot	1120
LightProbeGroup	220
LightProbes	258
LightProbesLegacy	197

Class	ID
LineRenderer	120
LODGroup	205
MasterServerInterface	155
Material	21
Mesh	43
Mesh3DSImporter	1005
MeshCollider	64
MeshFilter	33
MeshParticleEmitter	87
MeshRenderer	23
ModelImporter	1040
MonoBehaviour	114
MonoImporter	1035
MonoManager	116
MonoScript	115
Motion	207
MovieImporter	1044
MovieTexture	152
NamedObject	130
NativeFormatImporter	1034
NavMeshAgent	195
NavMeshAreas	126
NavMeshData	238
NavMeshObsolete	194
NavMeshObstacle	208
NavMeshSettings	196
NetworkManager	149
NetworkView	148
NewAnimationTrack	118
OcclusionArea	192
OcclusionPortal	41
OffMeshLink	191
ParticleAnimator	12
ParticleEmitter	88
ParticleRenderer	26
ParticleSystem	198
ParticleSystemRenderer	199
PhysicMaterial	134
Physics2DManager	51
Physics2DSettings	19
PhysicsManager	55

Class	ID
PhysicsMaterial2D	62
PhysicsUpdateBehaviour2D	246
Pipeline	17
PlatformEffector2D	251
PlayerSettings	129
PluginImporter	1050
PointEffector2D	250
PolygonCollider2D	60
Prefab	1001
PreloadData	150
PreviewAssetType	1108
ProceduralMaterial	185
ProceduralTexture	186
Projector	119
PVRImporter	1052
QualitySettings	47
RaycastCollider	140
RectTransform	224
ReflectionProbe	215
ReflectionProbes	216
Renderer	25
RenderSettings	104
RenderTexture	84
ResourceManager	147
Rigidbody	54
Rigidbody2D	50
RuntimeAnimatorController	93
SampleClip	271
SceneAsset	1032
SceneSettings	29
ScriptMapper	94
Shader	48
ShaderImporter	1007
ShaderVariantCollection	200
SkinnedCloth	163
SkinnedMeshRenderer	137
Skybox	45
SliderJoint2D	234
SparseTexture	171
SphereCollider	135
SpringJoint	145

Class	ID
SpringJoint2D	231
Sprite	213
SpriteCollider2D	66
SpriteRenderer	212
SpeedTreeImporter	1110
SpeedTreeWindAsset	228
SubstanceArchive	184
SubstanceImporter	1112
SurfaceEffector2D	252
TagManager	78
Terrain	218
TerrainCollider	154
TerrainData	156
TerrainInstance	210
TextAsset	49
TextMesh	102
TextScriptImporter	1031
Texture	27
Texture2D	28
Texture3D	117
TextureImporter	1006
TimeManager	5
TrailRenderer	96
Transform	4
Tree	193
TrueTypeFontImporter	1042
WebCamTexture	158
WheelCollider	146
WheelJoint2D	235
WindZone	182
WorldParticleCollider	76

Cache Server

Leave feedback

Unity has a completely automatic **Asset** pipeline. Whenever a source **Asset** like a .psd or an .fbx file is modified, Unity detects the change and automatically re-imports it. The imported data from the file is subsequently stored by Unity in an internal format.

This arrangement is designed to make the workflow as efficient and flexible as possible for an individual user. However, when working in a team, you may find that other users might keep making changes to Assets, all of which must be imported. Furthermore, Assets must be reimported when you switch between desktop and mobile build target platforms. The switch can therefore take a long time for large projects.

Caching the imported Assets data on the **Cache Server** drastically reduces the time it takes to import Assets.

Each Asset import is cached based on:

The Asset file itself
The import settings
Asset importer version
The current platform
If any of the above change, the Asset is re-imported. Otherwise, it is downloaded from the Cache Server.

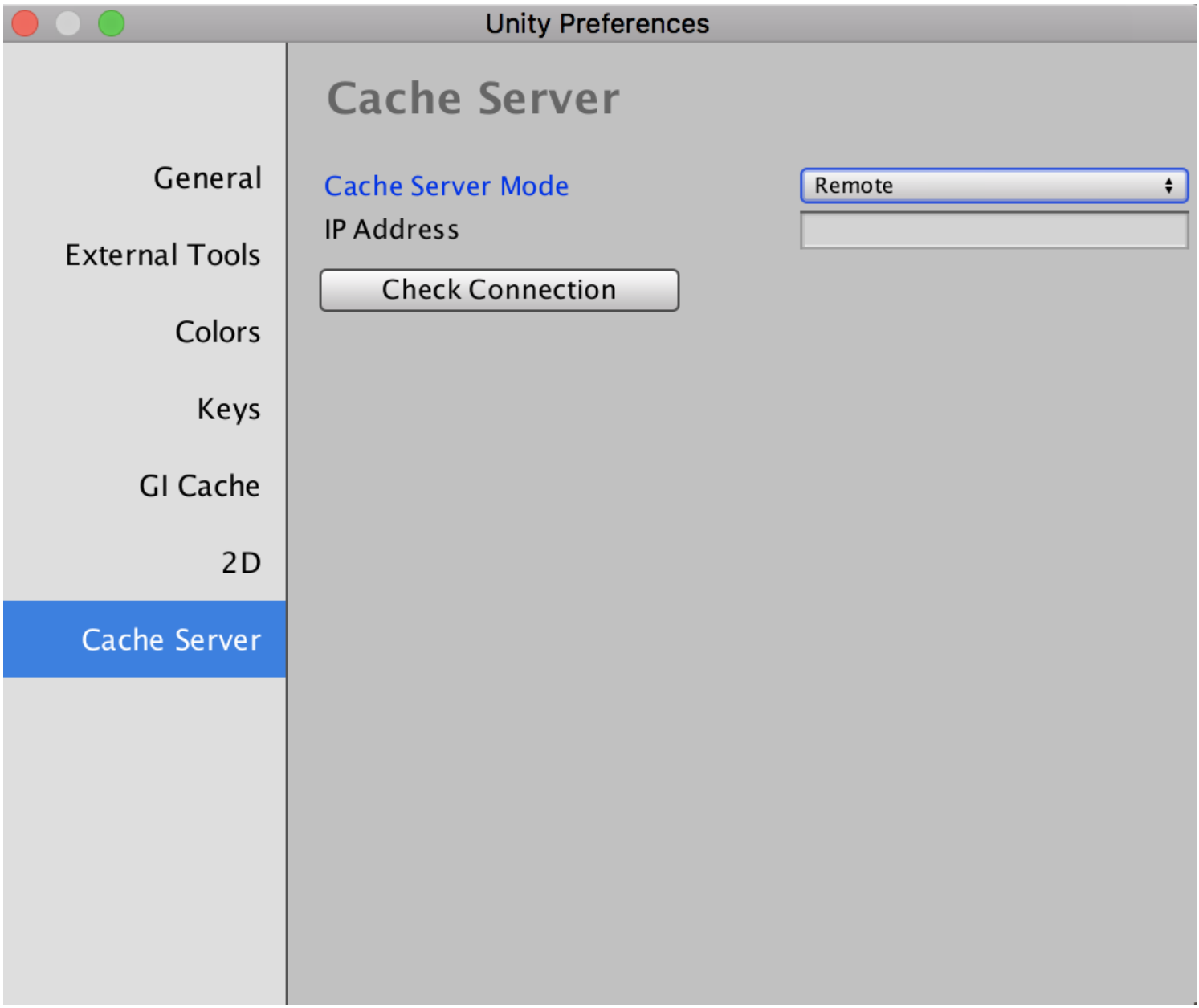
When you enable the Cache Server (see *How to set up a Cache Server as a user*, below), you can even share Asset imports across multiple projects (that is, the work of importing is done on one computer and the results are shared with others).

Note that once the Cache Server is set up, this process is completely automatic, so there are no additional workflow requirements. It will simply reduce the time it takes to import projects without getting in your way.

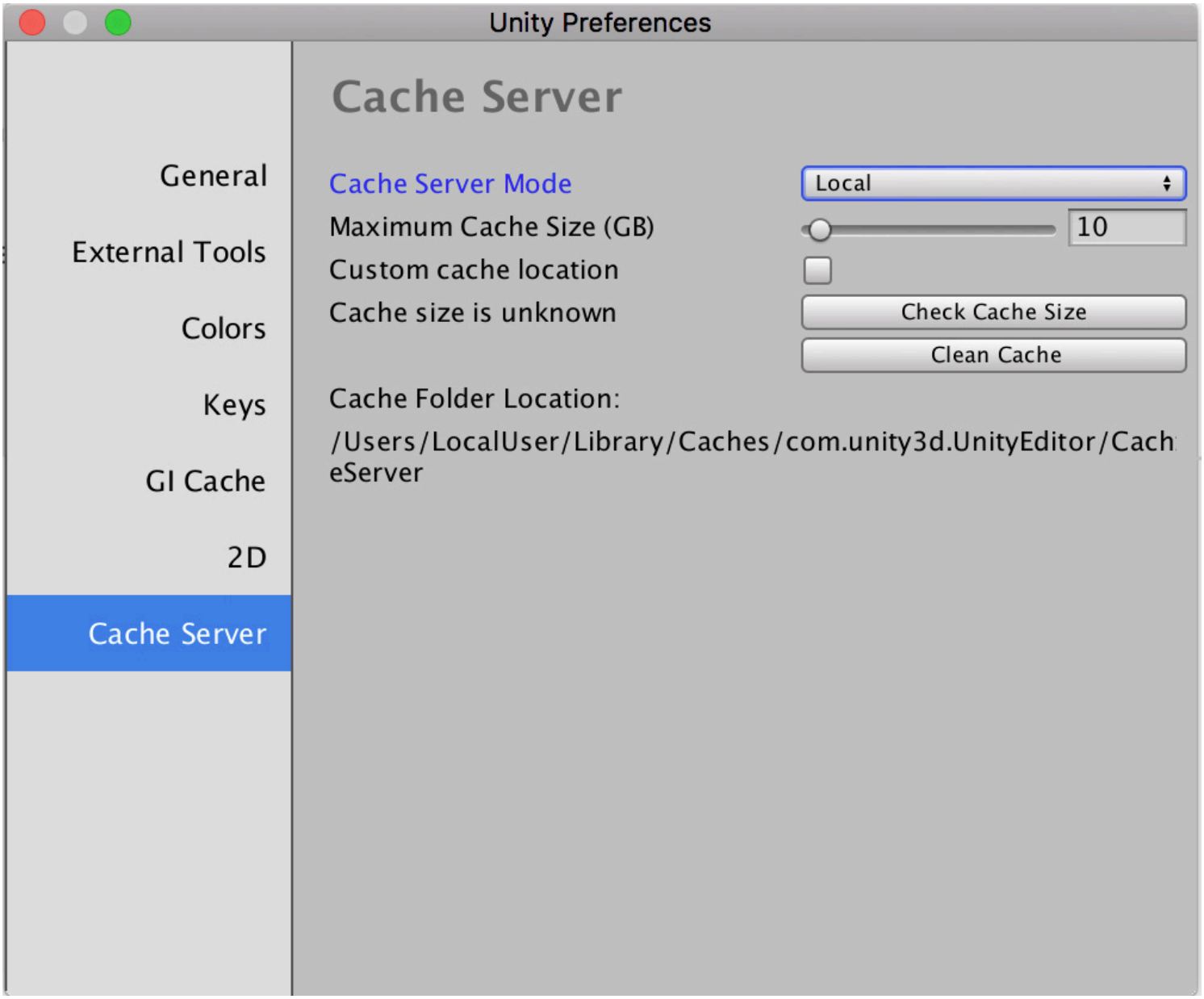
How to set up a Cache Server as a user

Setting up the Cache Server couldn't be easier. All you need to do is check Use Cache Server in the preferences and tell the local computer's **Unity Editor** where the **Cache Server** is.

The Cache Server settings can be found in **Unity** > **Preferences** on Mac OS X or **Edit** > **Preferences** on Windows and Linux.



To host the Cache Server on your local computer instead of a remote one, set **Cache Server Mode** to **Local**.



This setting allows you to easily configure a Cache Server on your local machine. Due to hard drive size limitations, it is recommended you host the Cache Server on a separate computer.

Property:	**Function:**
Cache Server Mode	Select the Cache Server Mode you wish to use. This setting allows you to disable use of the Cache Server, specify a remote server, or set up a Cache Server on your local computer.
Disabled (default)	Do not use a Cache Server.
Remote	Use a Cache Server hosted on a remote computer.
Local	Use a local Cache Server on this computer.
IP Address (Remote only)	Specify the IP address of the remote machine hosting the Cache Server.

Property:	Function:
Check Connection (Remote only)	Use this button to attempt to connect to the remote Cache Server.
Maximum Cache Size (GB) (Local only)	Specify a maximum size in gigabytes for the Cache Server on this computer's storage. The minimum size is 1GB. The maximum cache size is 200GB. The default is 10GB.
Custom cache location	Specify a location on disk to store the cache.
Check Cache Size (Local only)	Click this to find out how much storage the Local Cache Server is using. This operation can take some time to complete if you have a large project. The message "Cache size is unknown" is replaced with the cache size when it has finished running.
Clean Cache	Delete the contents of the cache.

Unity displays the following warning if you have a Local Cache Server with a custom location, and that location becomes unavailable:

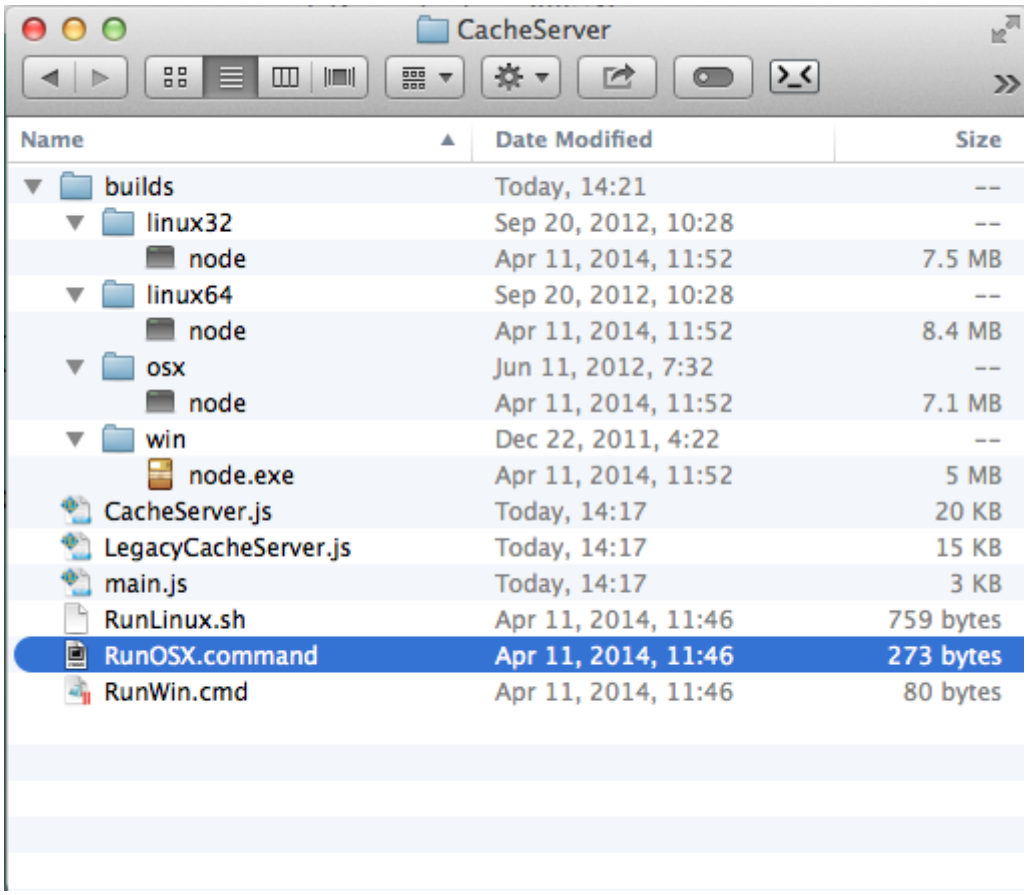
Local cache directory does not exist - please check that you can access the cache folder and are able to write to it

How to set up a Cache Server as an administrator

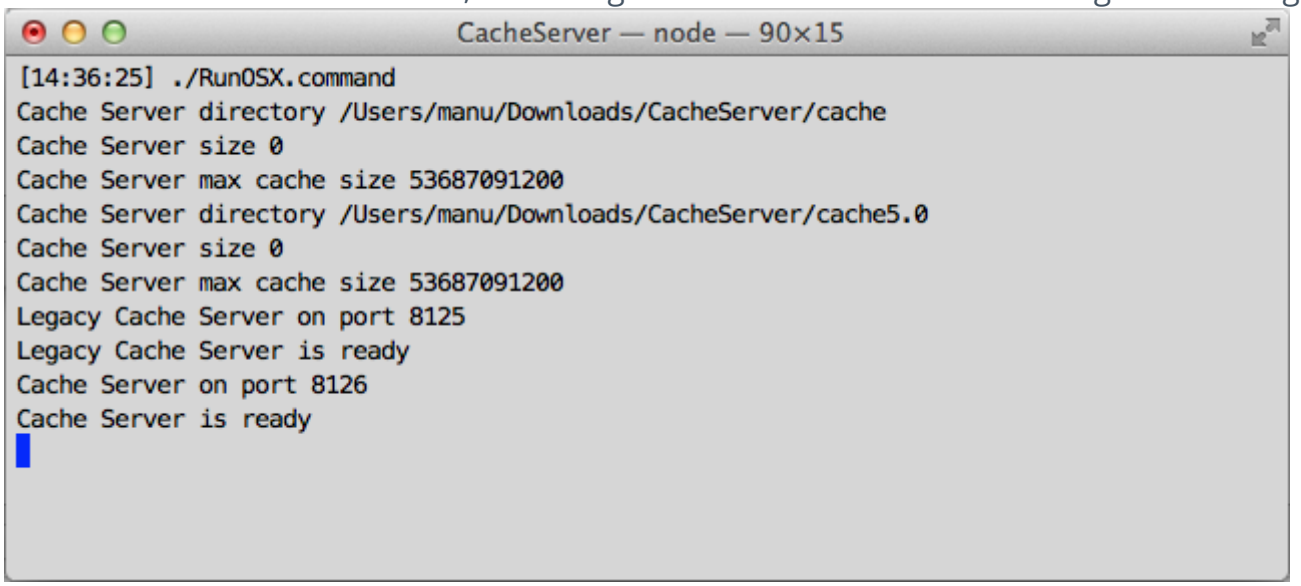
Admins need to set up the **Cache Server** computer that will host the cached Assets.

You need to:

Download the Cache Server. Go to the Download Archive page. Locate the Unity version you use and click on the Downloads button for your target server's operating system. Click the Cache Server link to start the download.
Unzip the file, after which you should see something like this:



Depending on your operating system, run the appropriate command script.
You will see a terminal window, indicating that the Cache Server is running in the background



The Cache Server needs to be on a reliable computer with very large storage (much larger than the size of the project itself, as there will be multiple versions of imported resources stored). If the hard disk becomes full the Cache Server could perform slowly.

Installing the Cache Server as a service

The provided `.sh` and `.cmd` **scripts** must be set up as a service on the server. The Cache Server can be safely killed and restarted at any time, since it uses atomic file operations.

New and legacy Cache Servers

Two Cache Server processes are started by default. The legacy Cache Server works with versions of Unity prior to version 5.0. The new Cache Server works with versions of Unity from 5.0 and up. See Cache Server configuration, below for details on configuring, enabling, and disabling the two different Cache Servers.

Cache Server configuration

If you simply start by executing the script, it launches the legacy Cache Server on port 8125 and the new Cache Server on port 8126. It also creates “cache” and “cache5.0” directories in the same directory as the script, and keep data in there. The cache directories are allowed to grow to up to 50 GB by default. You can configure the size and the location of the data using command line options, like this:

```
./RunOSX.command --path ~/mycachePath --size 2000000000
```

or

```
./RunOSX.command --path ~/mycachePath --port 8199 --nolegacy
```

You can configure the Cache Server by using the following command line options:

Use `--port` to specify the server port. This only applies to the new Cache Server. The default value is `8126`.
Use `--path` to specify the path of the cache location. This only applies to the new Cache Server. The default value is `./cache5.0`.
Use `--legacypath` to specify the path of the cache location. This only applies to the legacy Cache Server. The default value is `./cache`.
Use `--size` to specify the maximum cache size in bytes for both Cache Servers. Files that have not been used recently are automatically discarded when the cache size is exceeded.
Use `--nolegacy` to stop the legacy Cache Server starting. Otherwise, the legacy Cache Server is started on port `8125`.

Requirements for the computer hosting the Cache Server

For best performance there must be enough RAM to hold an entire imported project folder. In addition, it is best to have a computer with a fast hard drive and fast Ethernet connection. The hard drive should also have sufficient free space. On the other hand, the Cache Server has very low CPU usage.

One of the main distinctions between the Cache Server and **version control** is that its cached data can always be rebuilt locally. It is simply a tool for improving performance. For this reason it doesn’t make sense to use a Cache Server over the Internet. If you have a distributed team, you should place a separate Cache Server in each location.

The Cache Server runs optimally on a Linux or Mac OS X computer. The Windows file system is not particularly well-optimized for how the Cache Server stores data, and problems with file locking on Windows can cause issues that don’t occur on Linux or Mac OS X.

Cache Server FAQ

Will the size of my Cache Server database grow indefinitely as more and more resources get imported and stored?

The Cache Server removes Assets that have not been used for a period of time automatically (of course if those Assets are needed again, they are re-created on next usage).

Does the Cache Server work only with the Asset server?

The Cache Server is designed to be transparent to source/version control systems, so you are not restricted to using Unity's **Asset server**.

What changes cause the imported file to get regenerated?

When Unity is about to import an Asset, it generates an MD5 hash of all source data.

For a Texture, this consists of:

The source Asset: "myTexture.psd" file
The meta file: "myTexture.psd.meta" (Stores all importer settings)
The internal version number of the Texture Importer
A hash of version numbers of all AssetPostprocessors
If that hash is different from what is stored on the Cache Server, the Asset is reimported. Otherwise the cached version is downloaded. The client Unity Editor only pulls Assets from the server as they are needed - Assets don't get pushed to each project as they change.

How do I work with Asset dependencies?

The Cache Server does not handle dependencies. Unity's Asset pipeline does not deal with the concept of dependencies. It is built in such a way as to avoid dependencies between Assets. The AssetPostprocessor class is a common technique used to customize the Asset importer to fit your needs. For example, you might want to add MeshColliders to some **GameObjects** in an .fbx file based on their name or tag.

It is also easy to use `AssetPostprocessor` to introduce dependencies. For example you might use data from a text file next to the Asset to add additional **components** to the imported GameObjects. This is not supported in the Cache Server. If you want to use the Cache Server, you have to remove dependency on other Assets in the project folder. Since the Cache Server doesn't know anything about the dependency in your postprocessor, it does not know that anything has changed, and thus uses an old cached version of the Asset.

In practice there are plenty of ways you can do Asset postprocessing to work well with the Cache Server. You can use:

The Path of the imported Asset
Any import settings of the Asset
The source Asset itself, or any data generated from it passed to you in the Asset postprocessor.

Are there any issues when working with Materials?

Modifying Materials that already exist might cause trouble. When using the Cache Server, Unity validates that the references to Materials are maintained, but because no postprocessing calls are invoked, the contents of the Material cannot be changed when a model is imported through the Cache Server. Because of this, you might get different results when importing with and without Cache Server. It is best never to modify Materials that already exist on disk.

Are there any Asset types which are not cached by the server?

There are a few kinds of Asset data which the server doesn't cache. There isn't really anything to be gained by caching script files, so the server ignores them. Also, native files used by 3D modeling software (Autodesk® Maya®, Autodesk® 3ds Max®, etc) are converted to FBX using the application itself. The Asset server does not cache the native file nor the intermediate FBX file generated in the import process. However, it is possible to benefit from the server, by exporting files as FBX from the modeling software and then adding those to the Unity project.

Modifying Source Assets Through Scripting

Leave feedback

Automatic Instantiation

Usually when you want to make a modification to any sort of game **asset**, you want it to happen at runtime and you want it to be temporary. For example, if your character picks up an invincibility power-up, you might want to change the **shader** of the **material** for the player character to visually demonstrate the invincible state. This action involves modifying the material that's being used. This modification is not permanent because we don't want the material to have a different shader when we exit **Play Mode**.

However, it is possible in Unity to write **scripts** that will permanently modify a source asset. Let's use the above material example as a starting point.

To temporarily change the material's shader, we change the **shader** property of the **material component**.

```
private var invincibleShader = Shader.Find ("Specular");

function StartInvincibility {
    renderer.material.shader = invincibleShader;
}
```

When using this script and exiting Play Mode, the state of the **material** will be reset to whatever it was before entering Play Mode initially. This happens because whenever renderer.material is accessed, the material is automatically instantiated and the instance is returned. This instance is simultaneously and automatically applied to the renderer. So you can make any changes that your heart desires without fear of permanence.

Direct Modification

IMPORTANT NOTE

The method presented below will modify actual source asset files used within Unity. These modifications are not undoable. Use them with caution.

Now let's say that we don't want the material to reset when we exit play mode. For this, you can use renderer.sharedMaterial. The sharedMaterial property will return the actual asset used by this renderer (and maybe others).

The code below will permanently change the material to use the Specular shader. It will not reset the material to the state it was in before Play Mode.

```
private var invincibleShader = Shader.Find ("Specular");

function StartInvincibility {
    renderer.sharedMaterial.shader = invincibleShader;
}
```

As you can see, making any changes to a sharedMaterial can be both useful and risky. Any change made to a sharedMaterial will be permanent, and not undoable.

Applicable Class Members

The same formula described above can be applied to more than just materials. The full list of assets that follow this convention is as follows:

Materials: renderer.material and renderer.sharedMaterial
Meshes: meshFilter.**mesh** and meshFilter.sharedMesh
Physic Materials: **collider**.material and **collider**.sharedMaterial

Direct Assignment

If you declare a public variable of any above class: Material, Mesh, or Physic Material, and make modifications to the asset using that variable instead of using the relevant class member, you will not receive the benefits of automatic instantiation before the modifications are applied.

Assets that are not automatically instantiated

There are two different assets that are never automatically instantiated when modifying them.

Texture2D
TerrainData

Any modifications made to these assets through scripting are always permanent, and never undoable. So if you're changing your **terrain**'s heightmap through scripting, you'll need to account for instantiating and assigning values on your own. Same goes for Textures. If you change the **pixels** of a texture file, the change is permanent.

iOS and Android Notes

Texture2D assets are never automatically instantiated when modifying them in **iOS** and Android projects. Any modifications made to these assets through scripting are always permanent, and never undoable. So if you change the pixels of a texture file, the change is permanent.

Extending the Editor

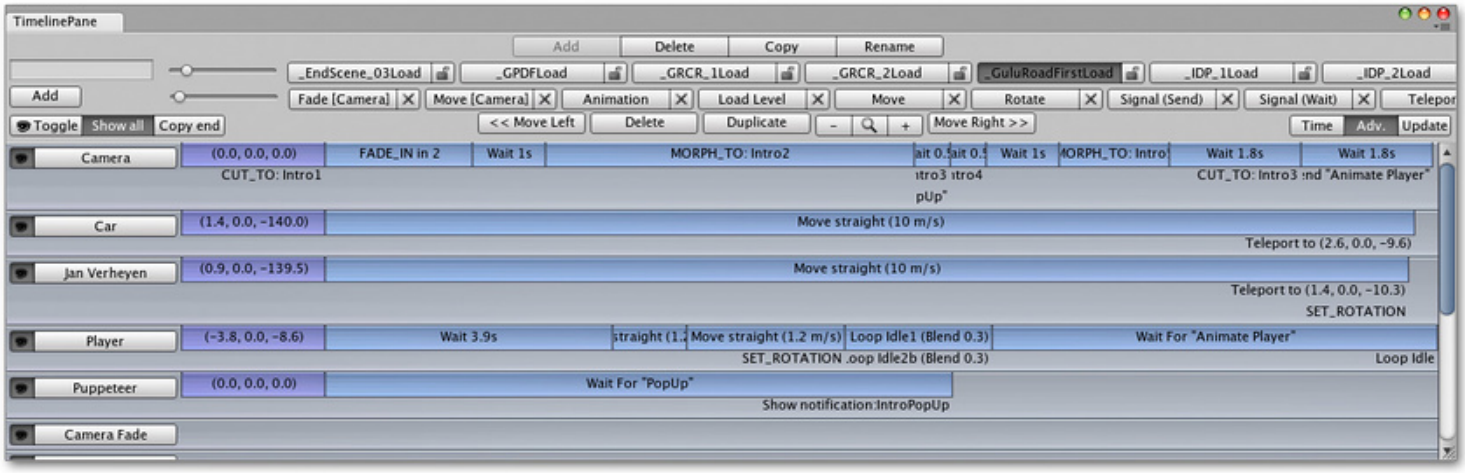
Leave feedback

Unity lets you extend the editor with your own custom **inspectors** and **Editor Windows** and you can define how properties are displayed in the inspector with custom **Property Drawers**. This section explains how to use these features.

Editor Windows

Leave feedback

You can create any number of custom windows in your app. These behave just like the **Inspector**, **Scene** or any other built-in ones. This is a great way to add a user interface to a sub-system for your game.



Custom Editor Interface by Serious Games Interactive used for scripting cutscene actions

Making a custom Editor Window involves the following simple steps:

Create a script that derives from EditorWindow.
Use code to trigger the window to display itself.
Implement the GUI code for your tool.

Derive From EditorWindow

In order to make your Editor Window, your script must be stored inside a folder called “Editor”. Make a class in this script that derives from EditorWindow. Then write your GUI controls in the inner OnGUI function.

```
using UnityEngine;
using UnityEditor;
using System.Collections;

public class Example : EditorWindow

    {
        void OnGUI () {
            // The actual window code goes here
           }
    }
```

MyWindow.js - placed in a folder called ‘Editor’ within your project.

Showing the window

In order to show the window on screen, make a menu item that displays it. This is done by creating a function which is activated by the **MenuItem** property.

The default behavior in Unity is to recycle windows (so selecting the menu item again would show existing windows. This is done by using the function EditorWindow.GetWindow Like this:

```
using UnityEngine;
using UnityEditor;
using System.Collections;

class MyWindow : EditorWindow {
    [MenuItem ("Window/My Window")]

    public static void  ShowWindow () {
        EditorWindow.GetWindow(typeof(MyWindow));
    }

    void OnGUI () {
        // The actual window code goes here
    }
}
```

Showing the MyWindow

This will create a standard, dockable editor window that saves its position between invocations, can be used in custom layouts, **etc**. To have more control over what gets created, you can use GetWindowWithRect

Implementing Your Window's GUI

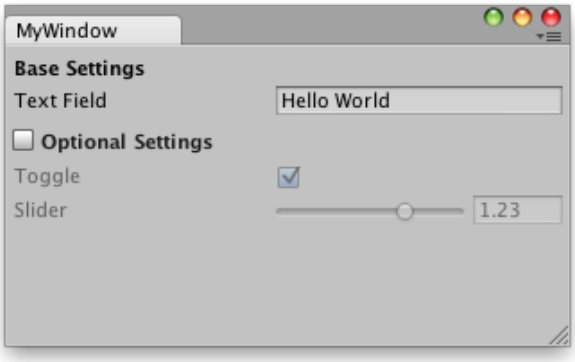
The actual contents of the window are rendered by implementing the OnGUI function. You can use the same UnityGUI classes you use for your ingame GUI (**GUI** and **GUILayout**). In addition we provide some additional GUI controls, located in the editor-only classes **EditorGUI** and **EditorGUILayout**. These classes add to the controls already available in the normal classes, so you can mix and match at will.

The following C# code shows how you can add GUI elements to your custom EditorWindow:

```
using UnityEditor;
using UnityEngine;

public class MyWindow : EditorWindow
{
    string myString = "Hello World";
    bool groupEnabled;
    bool myBool = true;
    float myFloat = 1.23f;

    // Add menu item named "My Window" to the Window menu
    [MenuItem("Window/My Window")]
    public static void ShowWindow()
    {
        //Show existing window instance. If one doesn't exist, make one.
        EditorWindow.GetWindow(typeof(MyWindow));
    }

    void OnGUI()
```

```
    {
        GUILayout.Label ("Base Settings", EditorStyles.boldLabel);
        myString = EditorGUILayout.TextField ("Text Field", myString);

        groupEnabled = EditorGUILayout.BeginToggleGroup ("Optional Settings", groupEnabled
            myBool = EditorGUILayout.Toggle ("Toggle", myBool);
            myFloat = EditorGUILayout.Slider ("Slider", myFloat, -3, 3);
        EditorGUILayout.EndToggleGroup ();
    }
}
```

This example results in a window which looks like this:



Custom Editor Window created using supplied example.

For more info, take a look at the example and documentation on the EditorWindow page.

Property Drawers

Leave feedback

Property Drawers can be used to customize the look of certain controls in the **Inspector window** by using attributes on your **scripts**, or by controlling how a specific `Serializable` class should look.

Property Drawers have two uses:

Customize the GUI of every instance of a Serializable class.
Customize the GUI of script members with custom **Property Attributes**.

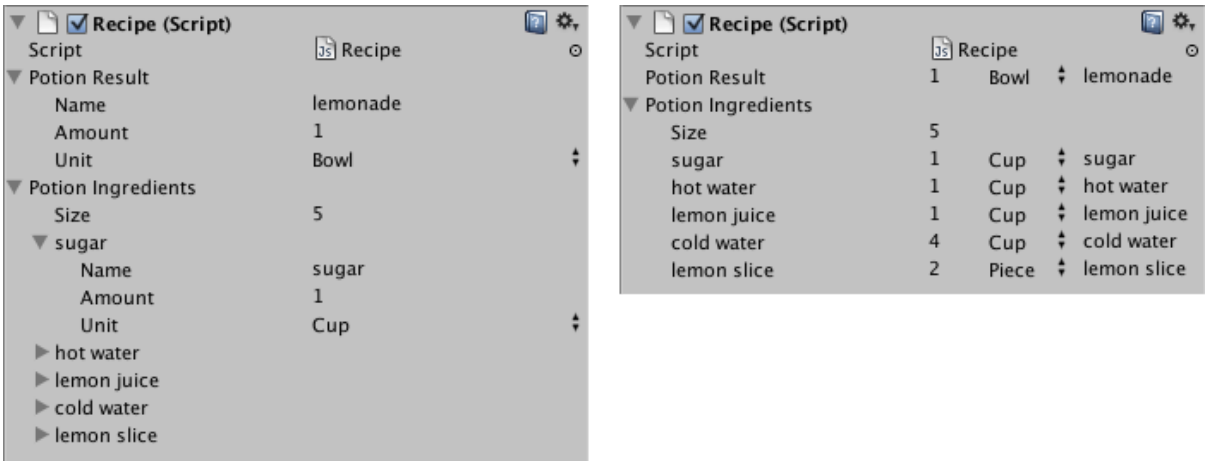
Customize the GUI of a Serializable class

If you have a custom **Serializable** class, you can use a **Property Drawer** to control how it looks in the **Inspector**. Consider the Serializable class Ingredient in the script examples below (**Note**: These are not editor scripts. Property attribute classes should be placed in a regular script file):

C# (example):

```
using System;
using UnityEngine;

enum IngredientUnit { Spoon, Cup, Bowl, Piece }

// Custom serializable class
[Serializable]
public class Ingredient
{
    public string name;
    public int amount = 1;
    public IngredientUnit unit;
}

public class Recipe : MonoBehaviour
{
    public Ingredient potionResult;
    public Ingredient[] potionIngredients;
}
```

Using a custom Property Drawer, every appearance of the Ingredient class in the Inspector can be changed. Compare the look of the Ingredient properties in the Inspector without and with a custom Property Drawer:



Class in the Inspector without (left) and with (right) custom Property Drawer.

You can attach the Property Drawer to a Serializable class by using the **CustomPropertyDrawer** attribute and pass in the type of the Serializable class that it's a drawer for.

C# (example):

```
using UnityEditor;
using UnityEngine;

// IngredientDrawer
[CustomPropertyDrawer(typeof(Ingredient))]
public class IngredientDrawer : PropertyDrawer
{
    // Draw the property inside the given rect
    public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)
    {
        // Using BeginProperty / EndProperty on the parent property means that
        // prefab override logic works on the entire property.
        EditorGUI.BeginProperty(position, label, property);

        // Draw label
        position = EditorGUI.PrefixLabel(position, GUIUtility.GetControlID(FocusType.Passive),

        // Don't make child fields be indented
        var indent = EditorGUI.indentLevel;
        EditorGUI.indentLevel = 0;

        // Calculate rects
        var amountRect = new Rect(position.x, position.y, 30, position.height);
        var unitRect = new Rect(position.x + 35, position.y, 50, position.height);
        var nameRect = new Rect(position.x + 90, position.y, position.width - 90, position.hei

        // Draw fields - passs GUIContent.none to each so they are drawn without labels
        EditorGUI.PropertyField(amountRect, property.FindPropertyRelative("amount"), GUIConten
        EditorGUI.PropertyField(unitRect, property.FindPropertyRelative("unit"), GUIContent.no
        EditorGUI.PropertyField(nameRect, property.FindPropertyRelative("name"), GUIContent.no

        // Set indent back to what it was
        EditorGUI.indentLevel = indent;

        EditorGUI.EndProperty();
```

```
    }
}
```

Customize the GUI of script members using Property Attributes

The other use of **Property Drawer** is to alter the appearance of members in a script that have custom **Property Attributes**. Say you want to limit floats or integers in your script to a certain range and show them as sliders in the **Inspector**. Using the built-in **PropertyAttribute** called **RangeAttribute** you can do just that:

C# (example):

```
// Show this float in the Inspector as a slider between 0 and 10
[Range(0f, 10f)]
float myFloat = 0f;
```

You can make your own **PropertyAttribute** as well. We'll use the code for the **RangeAttribute** as an example. The attribute must extend the **PropertyAttribute** class. If you want, your property can take parameters and store them as public member variables.

C# (example):

```
using UnityEngine;

public class MyRangeAttribute : PropertyAttribute
{
        readonly float min;
        readonly float max;

        void MyRangeAttribute(float min, float max)
        {
            this.min = min;
            this.max = max;
        }
}
```

Now that you have the attribute, you need to make a **Property Drawer** that draws properties that have that attribute. The drawer must extend the **PropertyDrawer** class, and it must have a **CustomPropertyDrawer** attribute to tell it which attribute it's a drawer for.

The property drawer class should be placed in an editor script, inside a folder called Editor.

C# (example):

```
using UnityEditor;
using UnityEngine;

// Tell the MyRangeDrawer that it is a drawer for properties with the MyRangeAttribute.
```

```
[CustomPropertyDrawer(typeof(MyRangeAttribute))]
public class RangeDrawer : PropertyDrawer
{
    // Draw the property inside the given rect
    void OnGUI(Rect position, SerializedProperty property, GUIContent label)
    {
        // First get the attribute since it contains the range for the slider
        MyRangeAttribute range = (MyRangeAttribute)attribute;

        // Now draw the property as a Slider or an IntSlider based on whether it's a float or
        if (property.propertyType == SerializedPropertyType.Float)
            EditorGUI.Slider(position, property, range.min, range.max, label);
        else if (property.propertyType == SerializedPropertyType.Integer)
            EditorGUI.IntSlider(position, property, (int) range.min, (int) range.max, label);
        else
            EditorGUI.LabelField(position, label.text, "Use MyRange with float or int.");
    }
}
```

Note that for performance reasons, EditorGUILayout functions are not usable with Property Drawers.

Custom Editors

Leave feedback

A key to increasing the speed of game creation is to create custom editors for commonly used **components**. For the sake of example, we'll use this very simple script that always keeps an object looking at a point. Add this script to your project, and place it onto a cube **gameobject** in your **scene**. The script should be named "LookAtPoint"

```
//C# Example (LookAtPoint.cs)
using UnityEngine;
public class LookAtPoint : MonoBehaviour
{
    public Vector3 lookAtPoint = Vector3.zero;

    void Update()
    {
        transform.LookAt(lookAtPoint);
    }
}
```

```
//JS Example (LookAtPoint.js)
#pragma strict
var lookAtPoint = Vector3.zero;
function Update()
{
    transform.LookAt(lookAtPoint);
}
```

This will keep an object oriented towards a world-space point. Currently this script will only become active in **play mode**, that is, when the game is running. When writing editor **scripts** it's often useful to have certain **scripts** execute during **edit mode** too, while the game is not running. You can do this by adding an ExecuteInEditMode attribute to it:

```
//C# Example (LookAtPoint.cs)
using UnityEngine;
[ExecuteInEditMode]
public class LookAtPoint : MonoBehaviour
{
    public Vector3 lookAtPoint = Vector3.zero;

    void Update()
    {
        transform.LookAt(lookAtPoint);
    }
}
```

```
//JS Example (LookAtPoint.js)
#pragma strict
@script ExecuteInEditMode()
var lookAtPoint = Vector3.zero;
function Update()
{
    transform.LookAt(lookAtPoint);
}
```

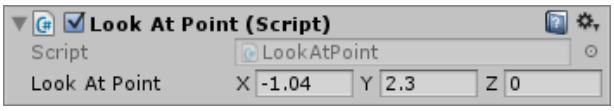
Now if you move the object which has this script around in the editor, or change the values of "Look At Point" in the **inspector** - even when not in play mode - the object will update its orientation correspondingly so it remains looking at the target point in world space.

Making a Custom Editor

The above demonstrates how you can get simple scripts running during edit-time, however this alone does not allow you to create your own editor tools. The next step is to create a **Custom Editor** for script we just created.

When you create a script in Unity, by default it inherits from MonoBehaviour, and therefore is a **Component** which can be placed on a game object. When placed on a game object, the Inspector displays a default interface for viewing and editing all public variables that can be shown - such as integers, floats, strings, Vector3's, **etc**.

Here's how the default inspector looks for our script above:



A default inspector with a public Vector3 field

A custom editor is a separate script which *replaces* this default layout with any editor controls that you choose.

To begin creating the custom editor for our LookAtPoint script, you should create another script with the same name, but with "Editor" appended. So for our example: "LookAtPointEditor".

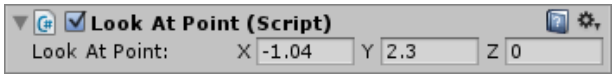
```
//c# Example (LookAtPointEditor.cs)
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(LookAtPoint))]
[CanEditMultipleObjects]
public class LookAtPointEditor : Editor
{
    SerializedProperty lookAtPoint;

    void OnEnable()
    {
        lookAtPoint = serializedObject.FindProperty("lookAtPoint");
    }

    public override void OnInspectorGUI()
    {
        serializedObject.Update();
```

```
        EditorGUILayout.PropertyField(lookAtPoint);
        serializedObject.ApplyModifiedProperties();
    }
}
```

```
//JS Example (LookAtPointEditor.js)
#pragma strict
@CustomEditor(LookAtPoint)
@CanEditMultipleObjects
class LookAtPointEditor extends Editor {

    var lookAtPoint : SerializedProperty;

    function OnEnable()
    {
        lookAtPoint = serializedObject.FindProperty("lookAtPoint");
    }

    function OnInspectorGUI()
    {
        serializedObject.Update();
        EditorGUILayout.PropertyField(lookAtPoint);
        serializedObject.ApplyModifiedProperties();
    }
}
```

This class has to derive from **Editor**. The **CustomEditor** attribute informs Unity which component it should act as an editor for. The **CanEditMultipleObjects** attribute tells Unity that you can select multiple objects with this editor and change them all at the same time.

The code in OnInspectorGUI is executed whenever Unity displays the editor in the Inspector. You can put any GUI code in here - it works just like OnGUI does for games, but is run inside the Inspector. Editor defines the target property that you can use to access the object being inspected. Here's what our custom inspector looks like:



It's not very interesting because all we have done so far is to recreate the Vector3 field, exactly like the default inspector shows us, so the result looks very similar (although the "Script" field is now not present, because we didn't add any inspector code to show it).

However now that you have control over how the inspector is displayed in an Editor script, you can use any code you like to lay out the inspector fields, allow the user to adjust the values, and even display graphics or other visual elements. In fact all of the inspectors you see within the Unity Editor including the more complex inspectors such as the **terrain** system and animation import settings, are all made using the same API that you have access to when creating your own custom Editors.

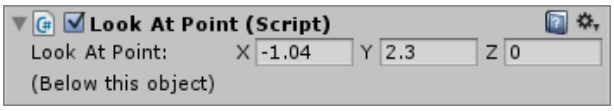
Here's a simple example which extends your editor script to display a message indicating whether the target point is above or below the gameobject:

```
//c# Example (LookAtPointEditor.cs)
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(LookAtPoint))]
[CanEditMultipleObjects]
public class LookAtPointEditor : Editor
{
    SerializedProperty lookAtPoint;

    void OnEnable()
    {
        lookAtPoint = serializedObject.FindProperty("lookAtPoint");
    }

    public override void OnInspectorGUI()
    {
        serializedObject.Update();
        EditorGUILayout.PropertyField(lookAtPoint);
        serializedObject.ApplyModifiedProperties();
        if (lookAtPoint.vector3Value.y > (target as LookAtPoint).transform.position.y)
        {
            EditorGUILayout.LabelField("(Above this object)");
        }
        if (lookAtPoint.vector3Value.y < (target as LookAtPoint).transform.position.y)
        {
            EditorGUILayout.LabelField("(Below this object)");
        }
    }
}
```

```
//JS Example (LookAtPointEditor.js)
#pragma strict
@CustomEditor(LookAtPoint)
@CanEditMultipleObjects
class LookAtPointEditor extends Editor {

    var lookAtPoint : SerializedProperty;

    function OnEnable()
    {
        lookAtPoint = serializedObject.FindProperty("lookAtPoint");
    }

    function OnInspectorGUI()
    {
        serializedObject.Update();
        EditorGUILayout.PropertyField(lookAtPoint);
        serializedObject.ApplyModifiedProperties();
```

```
        if (lookAtPoint.vector3Value.y > (target as LookAtPoint).transform.position.y)
        {
            EditorGUILayout.LabelField("(Above this object)");
        }
        if (lookAtPoint.vector3Value.y < (target as LookAtPoint).transform.position.y)
        {
            EditorGUILayout.LabelField("(Below this object)");
        }
    }
}
```

So now we have an new element to our inspector which prints a message showing if the target point is above or below the gameobject.



This is just scratching the surface of what you can do with Editor scripting. You have full access to all the IMGUI commands to draw any type of interface, including **rendering** scenes using a **camera** within editor windows.

Scene View Additions

You can add extra code to the **Scene View** by implementing an OnSceneGUI in your custom editor.

OnSceneGUI works just like OnInspectorGUI - except it gets run in the scene view. To help you make your own editing controls in the scene view, you can use the functions defined in Handles class. All functions in there are designed for working in 3D Scene views.

```
//C# Example (LookAtPointEditor.cs)
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(LookAtPoint))]
[CanEditMultipleObjects]
public class LookAtPointEditor : Editor
{
    SerializedProperty lookAtPoint;

    void OnEnable()
    {
        lookAtPoint = serializedObject.FindProperty("lookAtPoint");
    }

    public override void OnInspectorGUI()
    {
        serializedObject.Update();
        EditorGUILayout.PropertyField(lookAtPoint);
        if (lookAtPoint.vector3Value.y > (target as LookAtPoint).transform.position.y)
        {
            EditorGUILayout.LabelField("(Above this object)");
        }
        if (lookAtPoint.vector3Value.y < (target as LookAtPoint).transform.position.y)
        {
```

```
            EditorGUILayout.LabelField("(Below this object)");
        }


        serializedObject.ApplyModifiedProperties();
    }

    public void OnSceneGUI()
    {
        var t = (target as LookAtPoint);

        EditorGUI.BeginChangeCheck();
        Vector3 pos = Handles.PositionHandle(t.lookAtPoint, Quaternion.identity);
        if (EditorGUI.EndChangeCheck())
        {
            Undo.RecordObject(target, "Move point");
            t.lookAtPoint = pos;
            t.Update();
        }
    }
}
```

```
//JS Example (LookAtPointEditor.js)
#pragma strict
@CustomEditor(LookAtPointJS)
@CanEditMultipleObjects
class LookAtPointEditorJS extends Editor {

    var lookAtPoint : SerializedProperty;

    function OnEnable()
    {
        lookAtPoint = serializedObject.FindProperty("lookAtPoint");
    }

    function OnInspectorGUI()
    {
        serializedObject.Update();
        EditorGUILayout.PropertyField(lookAtPoint);
        serializedObject.ApplyModifiedProperties();
        if (lookAtPoint.vector3Value.y > (target as LookAtPointJS).transform.position.y)
        {
            EditorGUILayout.LabelField("(Above this object)");
        }
        if (lookAtPoint.vector3Value.y < (target as LookAtPointJS).transform.position.y)
        {
            EditorGUILayout.LabelField("(Below this object)");
        }
    }
```

```
    function OnSceneGUI()
    {
        var t : LookAtPointJS = (target as LookAtPointJS);

        EditorGUI.BeginChangeCheck();
        var pos = Handles.PositionHandle(t.lookAtPoint, Quaternion.identity);
        if (EditorGUI.EndChangeCheck())
        {
            Undo.RecordObject(target, "Move point");
            t.lookAtPoint = pos;
            t.Update();
        }
    }
}
```

If you want to put 2D GUI objects (GUI, EditorGUI and friends), you need to wrap them in calls to Handles.BeginGUI() and Handles.EndGUI().

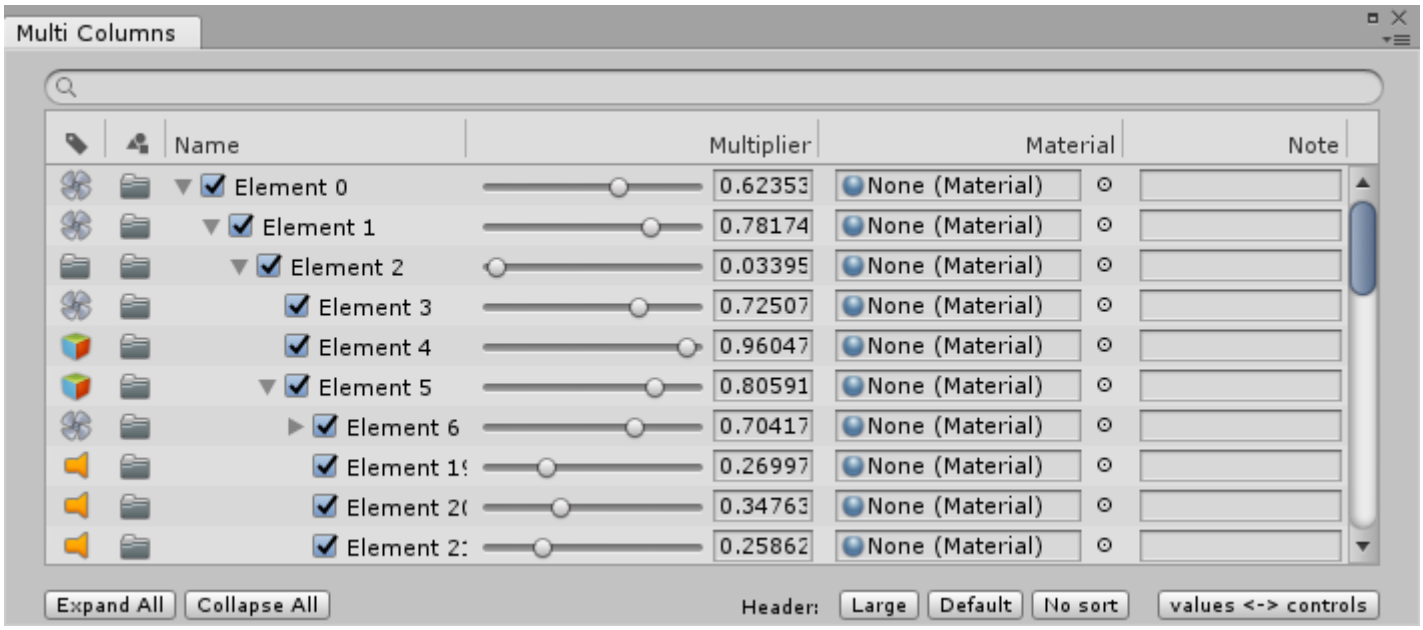
TreeView

Leave feedback

The information on this page assumes the reader has basic knowledge of IMGUI (Immediate Mode GUI) concepts. For information about IMGUI and customizing Editor windows, refer to Extending the Editor and the IMGUI Unity blog.

TreeView is an IMGUI control used to display hierarchical data that you can expand and collapse. Use TreeView to create highly customizable list views and multi-column tables for Editor windows, which you can use alongside other IMGUI controls and **components**.

See Unity Scripting API documentation on TreeView for information about the available TreeView API functions.



Example of a TreeView with a MultiColumnHeader and a SearchField.

Note that the TreeView is not a tree data model. You can construct TreeView using any tree data structure you prefer. This can be a C# tree model, or a Unity-based tree structure like the Transform hierarchy.

The **rendering** of the TreeView is handled by determining a list of expanded items called rows. Each row represents one `TreeViewItem`. Each `TreeViewItem` contains parent and children information, which is used by the TreeView to handle navigation (key and mouse input).

The TreeView has a single root `TreeViewItem` which is hidden and does not appear in the Editor. This item is the root of all other items.

Important classes and methods

The most important classes aside from the TreeView itself are TreeViewItem and TreeViewState.

TreeViewState (TreeViewState) contains state information that is changed when interacting with TreeView fields in the Editor, such as selection state, expanded state, navigation state, and scroll state. `TreeViewState` is the only state that is serializable. The TreeView itself is not serializable - it is reconstructed from the data that it represents when it is constructed or reloaded. Add the `TreeViewState` as a field in your `EditorWindow`-derived class to ensure that user-changed states are not lost when reloading **scripts** or entering Play mode (see

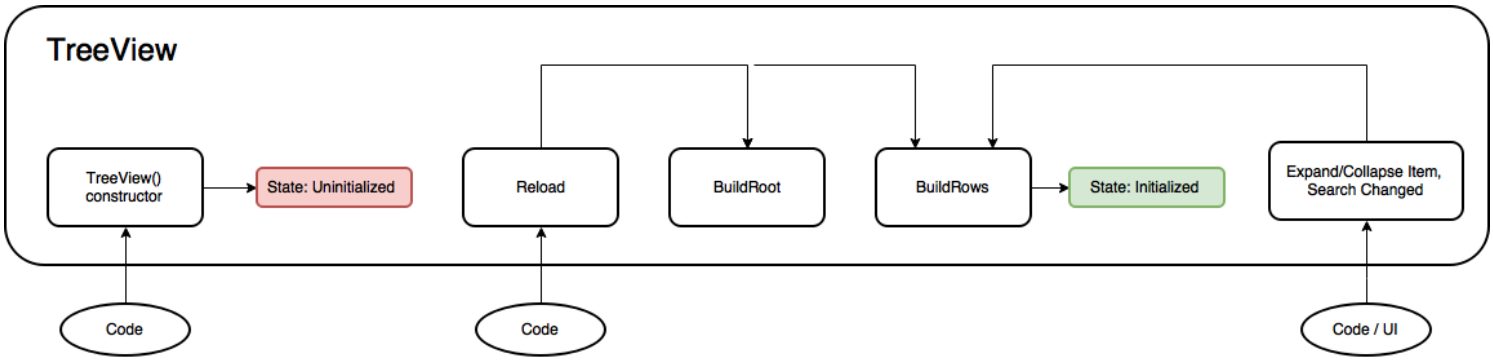
documentation on extending the Editor for information on how to do this). For an example of a class containing a `TreeViewState` field, see *Example 1: A simple TreeView* , below.

TreeViewItem (TreeViewItem) contains data about an individual TreeView item, and is used to build the representation of the tree structure in the Editor. Each `TreeViewItem` must be constructed with a unique integer ID (unique among all the items in the TreeView). The ID is used for finding items in the tree for the selection state, expanded state, and navigation. If the tree represents Unity objects, use GetInstanceID for each object as the ID for the `TreeViewItem`. The IDs are used in the `TreeViewState` to persist user-changed states (such as expanded items) when reloading scripts or entering Play mode in the Editor.

All `TreeViewItems` have a `depth` property, which indicates the visual indentation. See the *Initializing a TreeView* example below for more information.

BuildRoot (BuildRoot) is the single abstract method of the `TreeView` class that must be implemented to create a TreeView. Use this method to handle creating the root item of the tree. This is called every time Reload is called on the tree. For simple trees that use small data sets, create the entire tree of `TreeViewItems` under the root item in `BuildRoot`. For very large trees, it is not optimal to create the entire tree on every reload. In this situation, create the root and then override the `BuildRows` method to only create items for the current rows. For an example of `BuildRoot` in use, see *Example 1: A simple TreeView* below.

BuildRows (BuildRows) is a virtual method where the default implementation handles building the rows list based on the full tree created in `BuildRoot`. If only the root was created in `BuildRoot`, this method should be overridden to handle the expanded rows. See Initializing a TreeView, below, for more information.



This diagram summarizes the ordering and repetition of `BuildRoot` and `BuildRows` event methods during a TreeView's lifetime. Note that the `BuildRoot` method is called once every time `Reload` is called. `BuildRows` is called more often because it is called once on `Reload` (right after `BuildRoot`) and every time `TreeViewItem` is expanded or collapsed.

Initializing a TreeView

The TreeView is initialized when the `Reload` method is called from a TreeView object.

There are two ways to set up a TreeView:

Create the full tree - Create `TreeViewItem`s for all items in the tree model data. This is the default and requires less code to set up. The full tree is built when BuildRoot is called from a TreeView object.

Create only the expanded items - This approach requires you to override `BuildRows` to manually control the rows being shown, and `BuildRoot` is only used to create the root `TreeViewItem`. This approach scales best with large data sets or data that changes often.

Use the first approach for small data sets, or for data that does not change often. Use the second approach for large data sets, or data that changes often, because it is faster to only create only expanded items rather than a full tree.

There are three ways you can set up TreeViewItems:

Create `TreeViewItem`s with children, parent, and depths initialized from the start.

Create `TreeViewItem`s with parent and children and then use SetupDepthsFromParentsAndChildren to set the depths.

Create `TreeViewItem`s only with depth information and then use SetupDepthsFromParentsAndChildren to set the the parent and children references.

Examples

To view the Project and source code for the examples shown below, download TreeViewExamples.zip.

Example 1: A simple TreeView

To create a TreeView, create a class that extends the `TreeView` class and implement the abstract method `BuildRoot`. The following example creates a simple TreeView.

```
class SimpleTreeView : TreeView
{
    public SimpleTreeView(TreeViewState treeViewState)
        : base(treeViewState)
    {
        Reload();
    }

    protected override TreeViewItem BuildRoot ()
    {
        // BuildRoot is called every time Reload is called to ensure that TreeVi
```

```
        // are created from data. Here we create a fixed set of items. In a real
        // a data model should be passed into the TreeView and the items created

        // This section illustrates that IDs should be unique. The root item is
        // have a depth of -1, and the rest of the items increment from that.
        var root = new TreeViewItem {id = 0, depth = -1, displayName = "Root"};
        var allItems = new List<TreeViewItem>
        {
            new TreeViewItem {id = 1, depth = 0, displayName = "Animals"},
            new TreeViewItem {id = 2, depth = 1, displayName = "Mammals"},
            new TreeViewItem {id = 3, depth = 2, displayName = "Tiger"},
            new TreeViewItem {id = 4, depth = 2, displayName = "Elephant"},
            new TreeViewItem {id = 5, depth = 2, displayName = "Okapi"},
            new TreeViewItem {id = 6, depth = 2, displayName = "Armadillo"},
            new TreeViewItem {id = 7, depth = 1, displayName = "Reptiles"},
            new TreeViewItem {id = 8, depth = 2, displayName = "Crocodile"},
            new TreeViewItem {id = 9, depth = 2, displayName = "Lizard"},
        };

        // Utility method that initializes the TreeViewItem.children and .parent
        SetupParentsAndChildrenFromDepths (root, allItems);

        // Return root of the tree
        return root;
    }
}
```

In this example, depth information is used to build the TreeView. Finally, a call to `SetupDepthsFromParentsAndChildren` sets the parent and children data of the `TreeViewItem`s.

Note that there are two ways to set up the `TreeViewItem`: Set the parent and children directly, or use the `AddChild` method, as demonstrated in the following example:

```
protected override TreeViewItem BuildRoot()
{
    var root = new TreeViewItem      { id = 0, depth = -1, displayName = "Root"
    var animals = new TreeViewItem   { id = 1, displayName = "Animals" };
    var mammals = new TreeViewItem   { id = 2, displayName = "Mammals" };
    var tiger = new TreeViewItem     { id = 3, displayName = "Tiger" };
    var elephant = new TreeViewItem  { id = 4, displayName = "Elephant" };
    var okapi = new TreeViewItem     { id = 5, displayName = "Okapi" };
    var armadillo = new TreeViewItem { id = 6, displayName = "Armadillo" };
    var reptiles = new TreeViewItem  { id = 7, displayName = "Reptiles" };
```

```
    var croco = new TreeViewItem     { id = 8, displayName = "Crocodile" };
    var lizard = new TreeViewItem    { id = 9, displayName = "Lizard" };

    root.AddChild(animals);
    animals.AddChild(mammals);
    animals.AddChild(reptiles);
    mammals.AddChild(tiger);
    mammals.AddChild(elephant);
    mammals.AddChild(okapi);
    mammals.AddChild(armadillo);
    reptiles.AddChild(croco);
    reptiles.AddChild(lizard);

    SetupDepthsFromParentsAndChildren(root);

    return root;
}
```

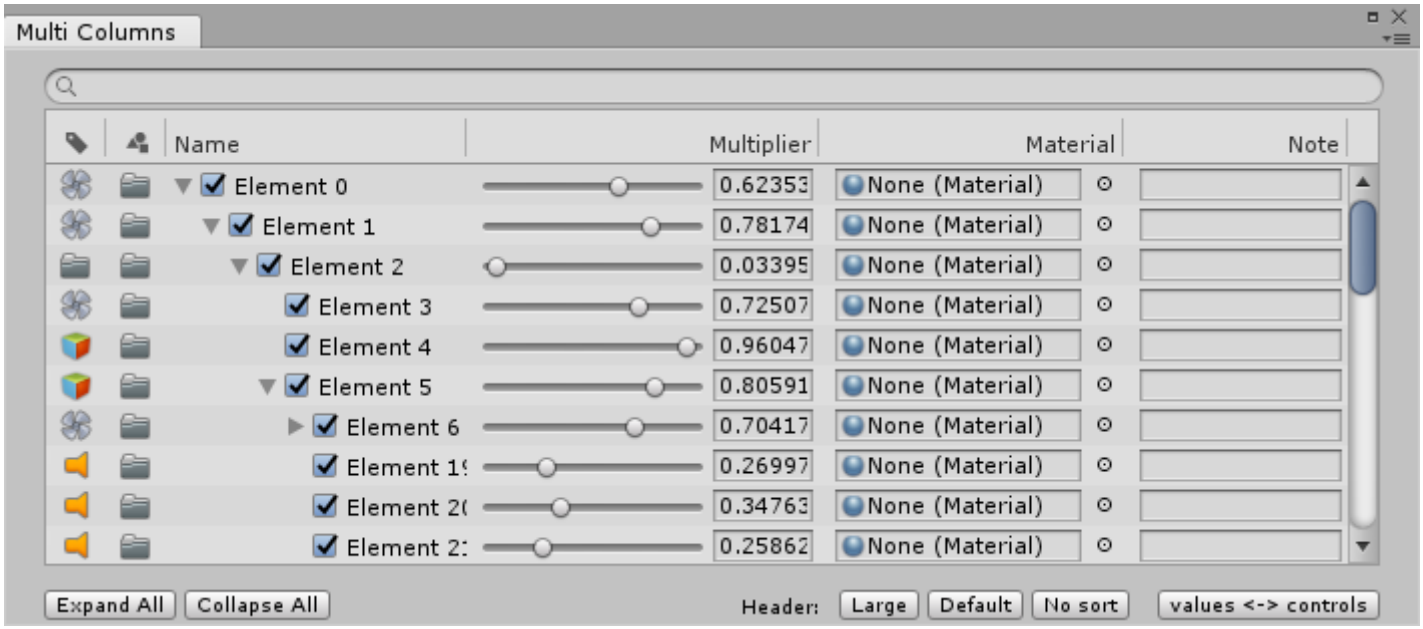
Alternative `BuildRoot` method for the `SimpleTreeView` class above

The following example shows the `EditorWindow` that contains the `SimpleTreeView`. TreeViews are constructed with a `TreeViewState` instance. The implementer of the TreeView should determine how this view state should be handled: whether its state should persist until the next session of Unity, or whether it should only preserve its state after scripts are reloaded (either when entering Play mode or recompiling scripts). In this example, the `TreeViewState` is serialized in the `EditorWindow`, ensuring the TreeView preserves its state when the Editor is closed and reopened.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEditor.IMGUI.Controls;

class SimpleTreeViewWindow : EditorWindow
{
    // SerializeField is used to ensure the view state is written to the window
    // layout file. This means that the state survives restarting Unity as long
    // is not closed. If the attribute is omitted then the state is still serial
    [SerializeField] TreeViewState m_TreeViewState;

    //The TreeView is not serializable, so it should be reconstructed from the t
    SimpleTreeView m_SimpleTreeView;

    void OnEnable ()
    {
```

```
        // Check whether there is already a serialized view state (state
        // that survived assembly reloading)
        if (m_TreeViewState == null)
            m_TreeViewState = new TreeViewState ();

        m_SimpleTreeView = new SimpleTreeView(m_TreeViewState);
    }

    void OnGUI ()
    {
        m_SimpleTreeView.OnGUI(new Rect(0, 0, position.width, position.height));
    }

    // Add menu named "My Window" to the Window menu
    [MenuItem ("TreeView Examples/Simple Tree Window")]
    static void ShowWindow ()
    {
        // Get existing open window or if none, make a new one:
        var window = GetWindow<SimpleTreeViewWindow> ();
        window.titleContent = new GUIContent ("My Window");
        window.Show ();
    }
}
```

Example 2: A multi-column TreeView



This example illustrates a multi-column TreeView that uses the MultiColumnHeader class.

`MultiColumnHeader` supports the following functionality: renaming items, multi-selection, reordering items and custom row content using normal IMGUI controls (such as sliders and object fields), sorting of columns, and the filtering and searching of rows.

This example creates a data model using the classes `TreeElement` and `TreeModel`. The TreeView fetches data from this "TreeModel". In this example, the `TreeElement` and `TreeModel` classes have been built in to demonstrate the features of the TreeView class. These classes have been included in the TreeView Examples Project (TreeViewExamples.zip). The example also shows how the tree model structure is serialized to a ScriptableObject and saved in an Asset.

```
[Serializable]
//The TreeElement data class is extended to hold extra data, which you can show
internal class MyTreeElement : TreeElement
{
    public float floatValue1, floatValue2, floatValue3;
    public Material material;
    public string text = "";
    public bool enabled = true;

    public MyTreeElement (string name, int depth, int id) : base (name, depth, i
    {
        floatValue1 = Random.value;
        floatValue2 = Random.value;
        floatValue3 = Random.value;
    }
}
```

The following ScriptableObject class ensures that data persists in an Asset when the tree is serialized.

```
[CreateAssetMenu (fileName = "TreeDataAsset", menuName = "Tree Asset", order = 1
public class MyTreeAsset : ScriptableObject
{
    [SerializeField] List<MyTreeElement> m_TreeElements = new List<MyTreeElement

    internal List<MyTreeElement> treeElements
    {
        get { return m_TreeElements; }
        set { m_TreeElements = value; }
    }
}
```

Construction of the MultiColumnTreeView class

The following example shows snippets of the class `MultiColumnTreeView`, which illustrates how the multi column GUI is achieved. Find full source code in the TreeView Examples Project (TreeViewExamples.zip).

```
public MultiColumnTreeView (TreeViewState state,
                            MultiColumnHeader multicolumnHeader,
                            TreeModel<MyTreeElement> model)
                            : base (state, multicolumnHeader, model)
{
    // Custom setup
    rowHeight = 20;
    columnIndexForTreeFoldouts = 2;
    showAlternatingRowBackgrounds = true;
    showBorder = true;
    customFoldoutYOffset = (kRowHeights - EditorGUIUtility.singleLineHeight) * 0
    extraSpaceBeforeIconAndLabel = kToggleWidth;
    multicolumnHeader.sortingChanged += OnSortingChanged;

    Reload();
}
```

The custom changes in the code sample above make the following adjustments:

`rowHeight = 20`: Change the default height (which is based on EditorGUIUtility.singleLineHeight's 16 points) to 20, to add more room for GUI controls.

`columnIndexForTreeFoldouts = 2`: In the example, the fold-out arrows are shown in the third column because this value is set to 2 (see image above). If this value is not changed, the fold-outs are rendered in the first column, because "columnIndexForTreeFoldouts" is 0 by default.

`showAlternatingRowBackgrounds = true`: Enable alternating row background colors, so that each row is distinct.

`showBorder = true`: Render the TreeView with a margin around it, so that a thin border is shown to delimit it from the rest of the content

`customFoldoutYOffset = (kRowHeights - EditorGUIUtility.singleLineHeight) * 0.5f`: Center fold-outs vertically in the row - see Customizing the GUI below.

`extraSpaceBeforeIconAndLabel = 20`: Make space before the tree labels so the toggle button is shown.

`multicolumnHeader.sortingChanged += OnSortingChanged`: Assign a method to the event to detect when the sorting changes in the header component (when the header column is clicked), so that the rows of the TreeView change to reflect the sorting state.

Customizing the GUI

If the default RowGUI handling is used, the TreeView looks like the `SimpleTreeView` example above, with only fold-outs and a label. When using multiple data values for each item, you must override the RowGUI method to visualize these values.

```
protected override void RowGUI (RowGUIArgs args)
```

The following code sample is the argument structure of the `RowGUIArgs` struct.

```
protected struct RowGUIArgs
{
    public TreeViewItem item;
    public string label;
    public Rect rowRect;
    public int row;
    public bool selected;
    public bool focused;
    public bool isRenaming;

    public int GetNumVisibleColumns ()
    public int GetColumn (int visibleColumnIndex)
    public Rect GetCellRect (int visibleColumnIndex)
}
```

You can extend the `TreeViewItem` and add additional user data (which creates a class that derives from `TreeViewItem`). You can then use this user data in the RowGUI callback. An example of this is provided below. See `override void RowGUI` - this example casts the input item to `TreeViewItem<MyTreeElement>`.

There are three methods which are related to column handling: GetNumVisibleColumns, GetColumn, and GetCellRect. You can only call these when the TreeView is constructed with a MultiColumnHeader, otherwise an exception is thrown.

```
protected override void RowGUI (RowGUIArgs args)
{
```

```
    var item = (TreeViewItem<MyTreeElement>) args.item;

    for (int i = 0; i < args.GetNumVisibleColumns (); ++i)
    {
        CellGUI(args.GetCellRect(i), item, (MyColumns)args.GetColumn(i), ref arg
    }
}
```

```
void CellGUI (Rect cellRect, TreeViewItem<MyTreeElement> item, MyColumns column,
{
    // Center the cell rect vertically using EditorGUIUtility.singleLineHeight.
// This makes it easier to place controls and icons in the cells.
    CenterRectUsingSingleLineHeight(ref cellRect);

    switch (column)
    {

        case MyColumns.Icon1:

            // Draw custom texture
GUI.DrawTexture(cellRect, s_TestIcons[GetIcon1Index(item)], ScaleMode.ScaleToFit
            break;

        case MyColumns.Icon2:

//Draw custom texture
            GUI.DrawTexture(cellRect, s_TestIcons[GetIcon2Index(item)], ScaleMod
            break;

        case MyColumns.Name:

            // Make a toggle button to the left of the label text
            Rect toggleRect = cellRect;
            toggleRect.x += GetContentIndent(item);
            toggleRect.width = kToggleWidth;
            if (toggleRect.xMax < cellRect.xMax)
                item.data.enabled = EditorGUI.Toggle(toggleRect, item.data.enabl

            // Default icon and label
            args.rowRect = cellRect;
            base.RowGUI(args);
            break;
```

```
        case MyColumns.Value1:

// Show a Slider control for value 1
            item.data.floatValue1 = EditorGUI.Slider(cellRect, GUIContent.none,
            break;

        case MyColumns.Value2:

// Show an ObjectField for materials
            item.data.material = (Material)EditorGUI.ObjectField(cellRect, GUICo
                                        typeof(Material), false);
            break;

        case MyColumns.Value3:

// Show a TextField for the data text string
            item.data.text = GUI.TextField(cellRect, item.data.text);
            break;
    }
}
```

TreeView FAQ

Q: In my TreeView subclass, I have the functions BuildRoot and RowGUI. Is `RowGUI` called for every `TreeViewItem` that got added in the build function, or only for items that are visible on screen in the scroll view?

A: `RowGUI` is only called for the items visible on screen. For example, if you have 10,000 items, only the 20 visible items on screen have their `RowGUI` called.

Q: Can I get the indices of the rows that are visible on the screen?

A: Yes. Use the method GetFirstAndLastVisibleRows.

Q: Can I get the list of rows that are built in BuildRows?

A: Yes. Use the method GetRows.

Q: Is it mandatory for any of the overridden functions to call `base.Method`?

A: Only if the method has a default behavior you want to extend.

Q: I just want to make list of items (not a tree). Do I have to create the root?

A: Yes, you should always have a root. You can create the root item and set `root.children = rows` for fast setup.

Q: I've added a Toggle to my row - why doesn't the selection jump to that row when I click on it?

A: By default, the row is only selected if the mouse down is not consumed by the contents of the row. Here, your Toggle consumes the event. To fix this, use the method SelectionClick before your Toggle button is called.

Q: Are there methods I can use before or after all `RowGUI` methods are called?

A: Yes. See API documentation on BeforeRowsGUI and AfterRowsGUI.

Q: Is there a simple way to return key focus to the TreeView from API? If I select a FloatField in my row, the row selection becomes gray. How do I make it blue again?

A: The blue color indicates which row currently has key focus. Because the FloatField has focus, the TreeView loses focus, so this is the intended behavior. Set `GUIUtility.keyboardControl = treeViewControlID` when needed.

Q: How do I convert from `id` to a `TreeViewItem`?

A: Use either FindItem or FindRows.

Q: How do I receive a callback when a user changes their selection in the TreeView?

A: Override the SelectionChanged method (other useful callbacks: DoubleClickedItem, and ContextClickedItem).

Running Editor Script Code on Launch

Leave feedback

Sometimes, it is useful to be able to run some editor script code in a project as soon as Unity launches without requiring action from the user. You can do this by applying the **InitializeOnLoad** attribute to a class which has a **static constructor**. A static constructor is a function with the same name as the class, declared static and without a return type or parameters (see here for more information):-

```
using UnityEngine;
using UnityEditor;

[InitializeOnLoad]
public class Startup {
    static Startup()
    {
        Debug.Log("Up and running");
    }
}
```

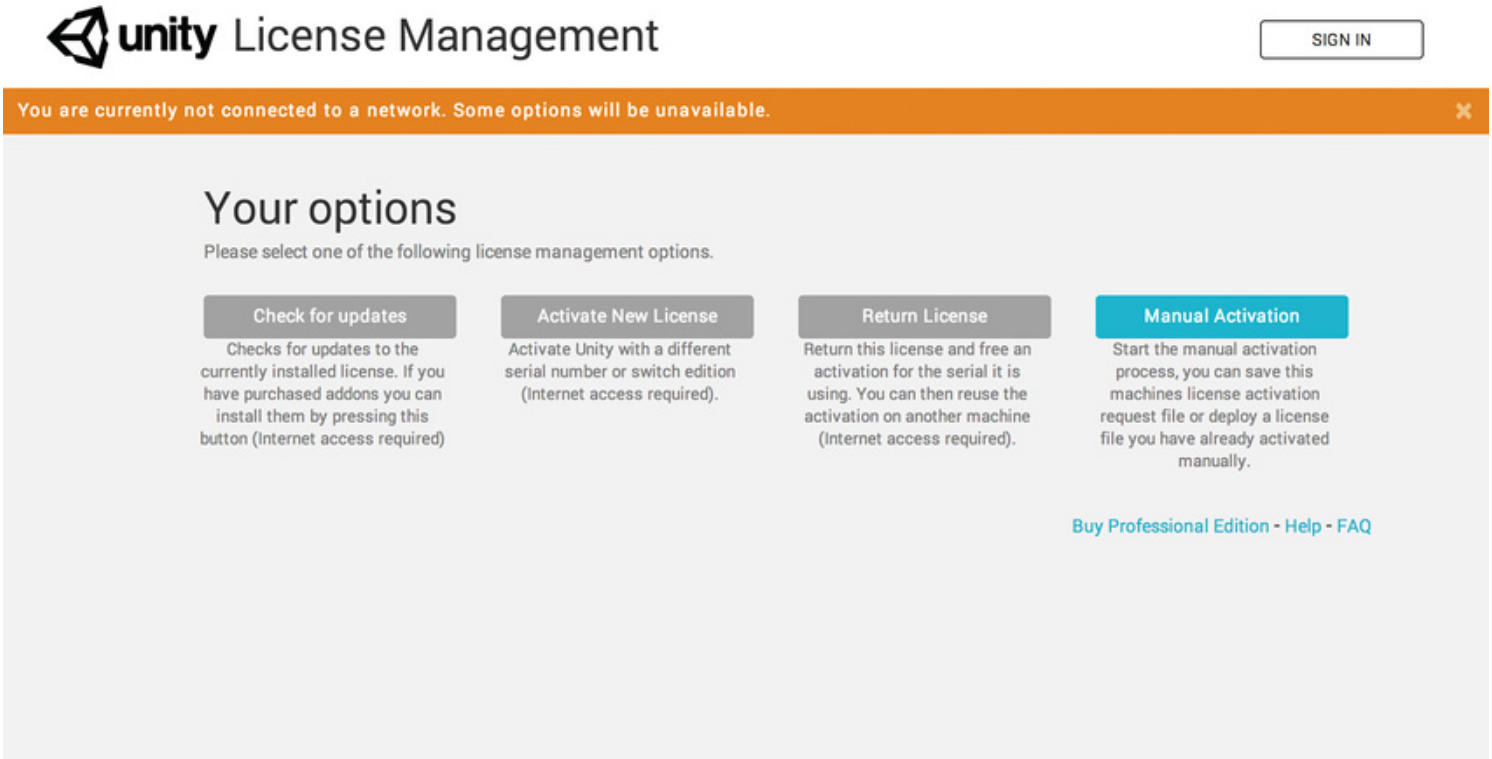
A static constructor is always guaranteed to be called before any static function or instance of the class is used, but the InitializeOnLoad attribute ensures that it is called as the editor launches.

An example of how this technique can be used is in setting up a regular callback in the editor (its “frame update”, as it were). The EditorApplication class has a delegate called update which is called many times a second while the editor is running. To have this delegate enabled as the project launches, you could use code like the following:-

```
using UnityEditor;
using UnityEngine;

[InitializeOnLoad]
class MyClass
{
    static MyClass ()
    {
        EditorApplication.update += Update;
    }

    static void Update ()
    {
        Debug.Log("Updating");
    }
}
```

Licenses and Activation

Leave feedback



Unity > Manage License

This section covers license activation and managing your Unity license. See the License FAQ page for help on common licensing questions.

2017–09–06 Page amended with limited editorial review

License activation updated in Unity 2017.2

Online Activation

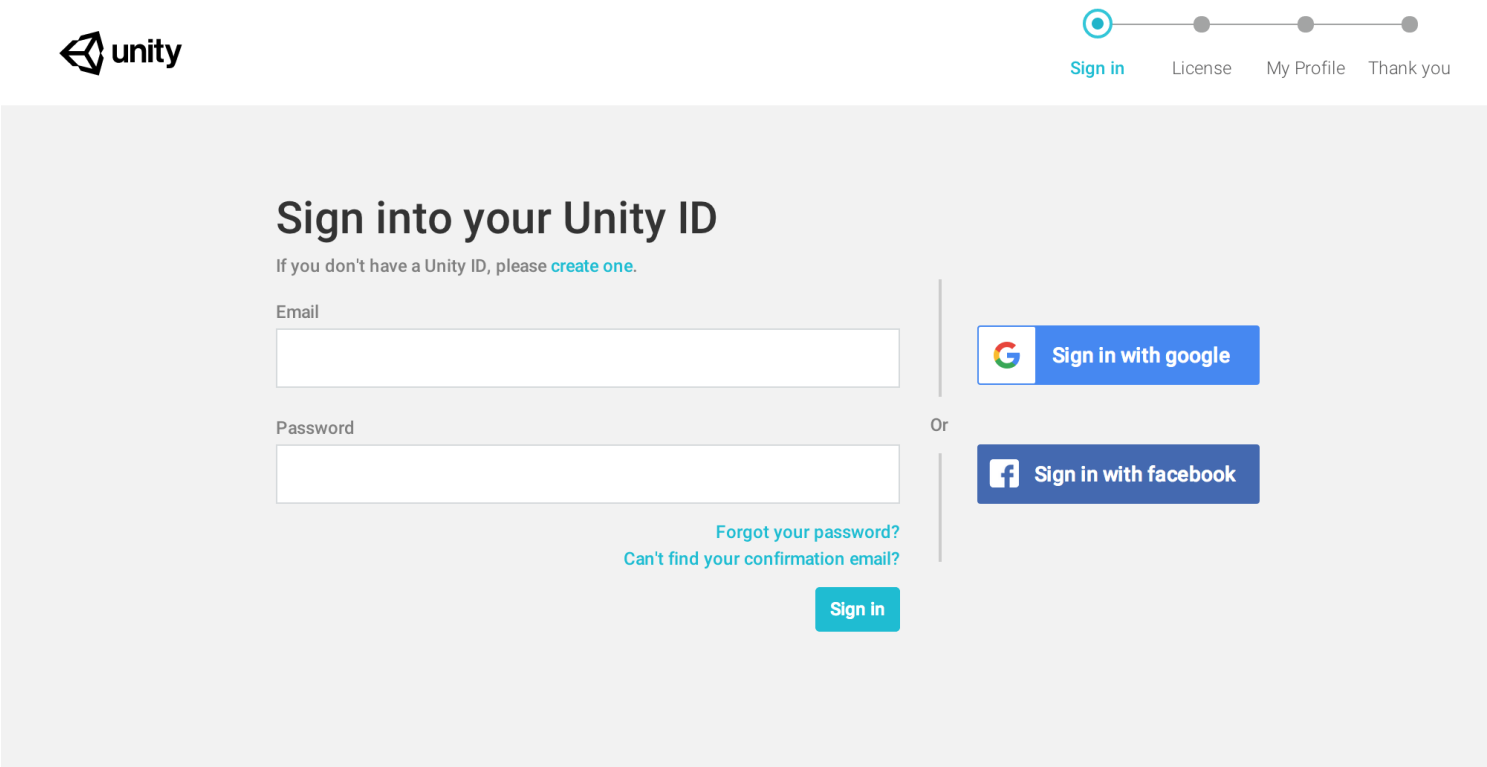
Leave feedback

Online activation is the easiest and fastest way to get up and running with Unity. Below is a step-by-step guide on how to activate Unity online.

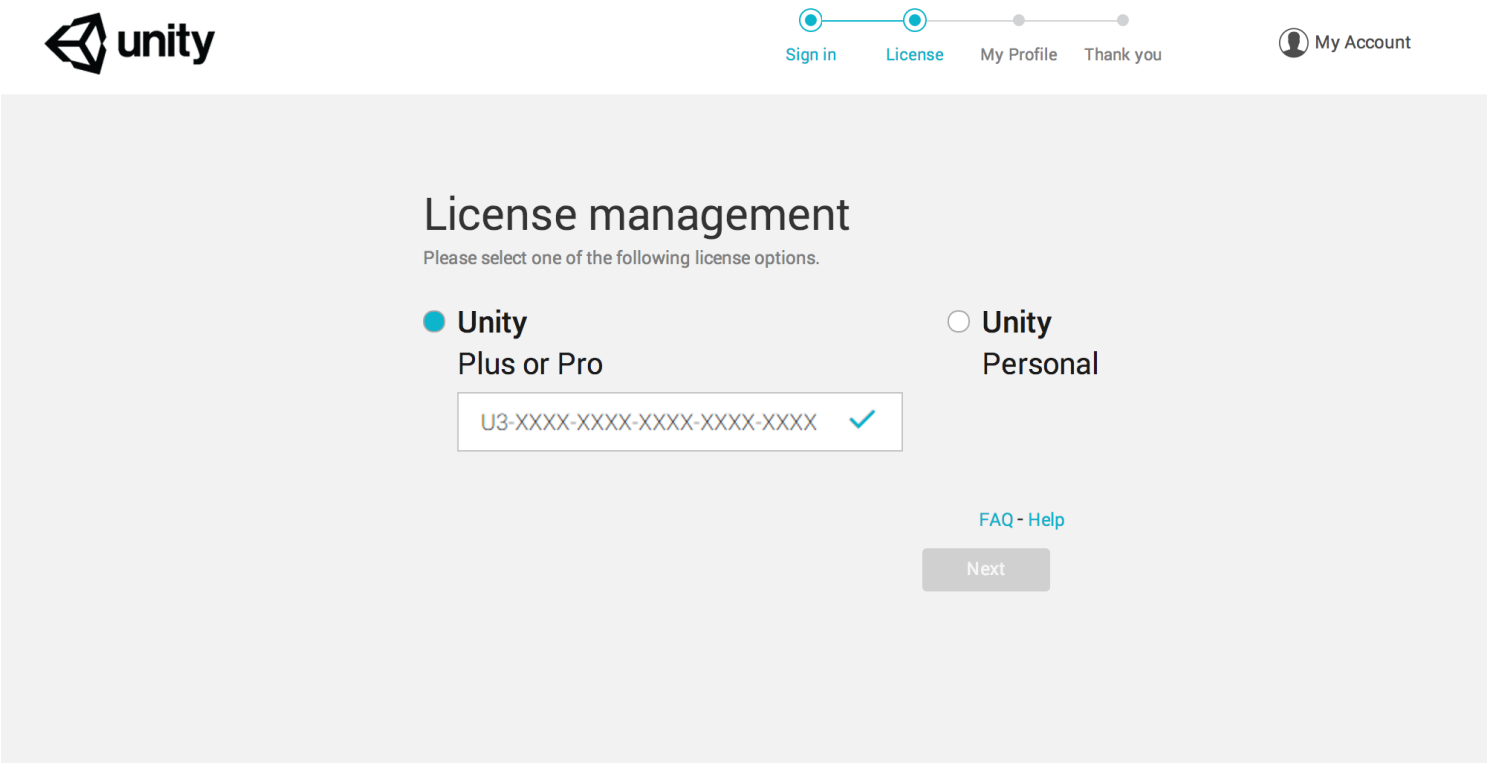
Download and install the Unity Editor. The latest version of Unity can be found here.

Fire up the Editor from your Applications folder on OS X or the shortcut in the Start Menu on Windows.

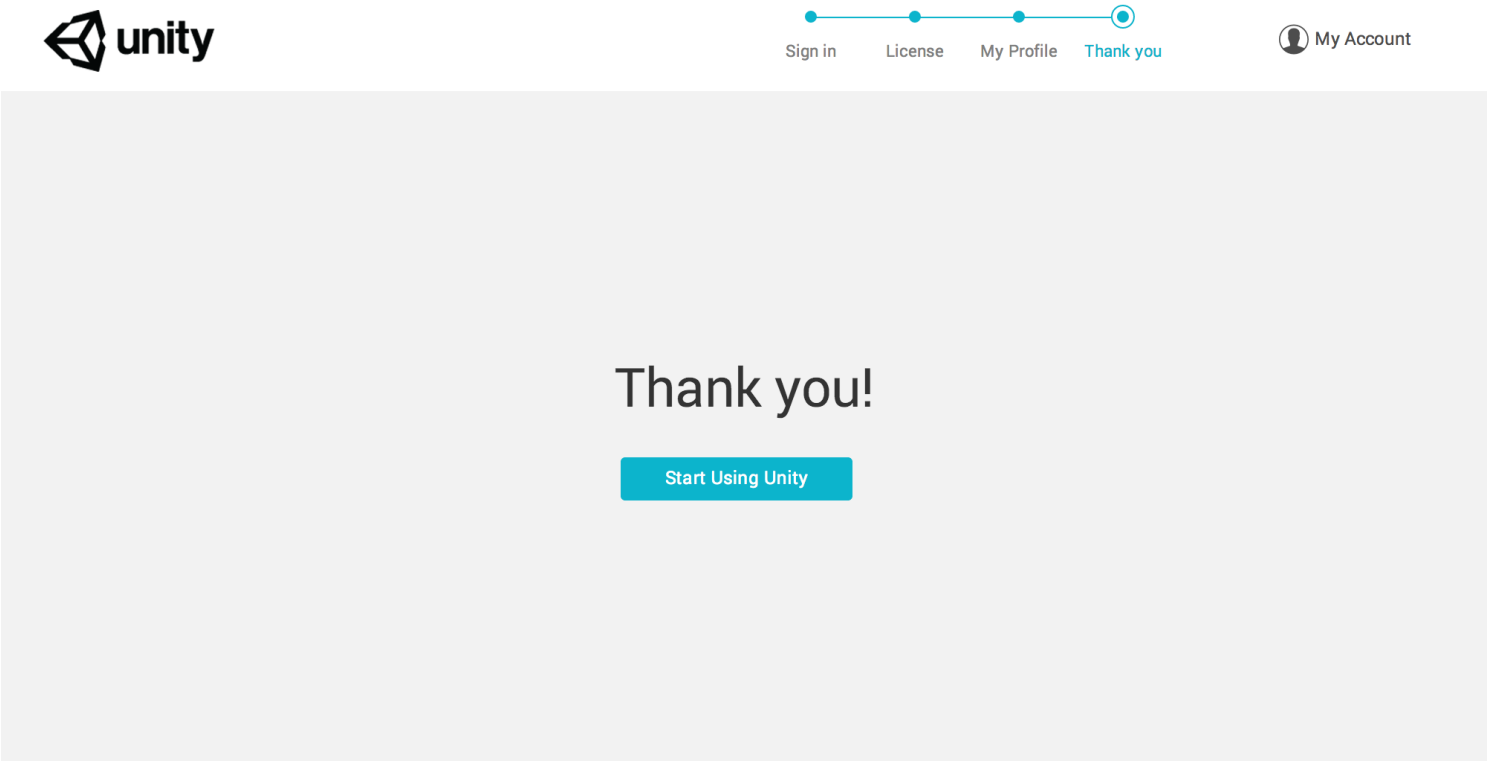
Firstly, you will encounter the 'Unity Account' window. Here you will need to enter your Unity Developer Network, Google or Facebook account credentials. (If you don't have an existing Unity account or have forgotten your password, simply click the respective 'create one' and 'Forgot your password?' links. Follow the onscreen prompts to create or retrieve your account.) Once your credentials are entered, you can proceed by clicking 'Sign in'.



You will be faced with a window titled 'License management'. Select the version of Unity you wish to activate and click 'Next'.



You will now be able to proceed to the Unity Editor by clicking the 'Start using Unity' button.



For any further assistance, contact **support@unity3d.com**.

2017–09–06 Page amended with limited editorial review

License activation updated in Unity 2017.2

Offline / Manual Activation

Leave feedback

If Online Activation fails, this could be because:

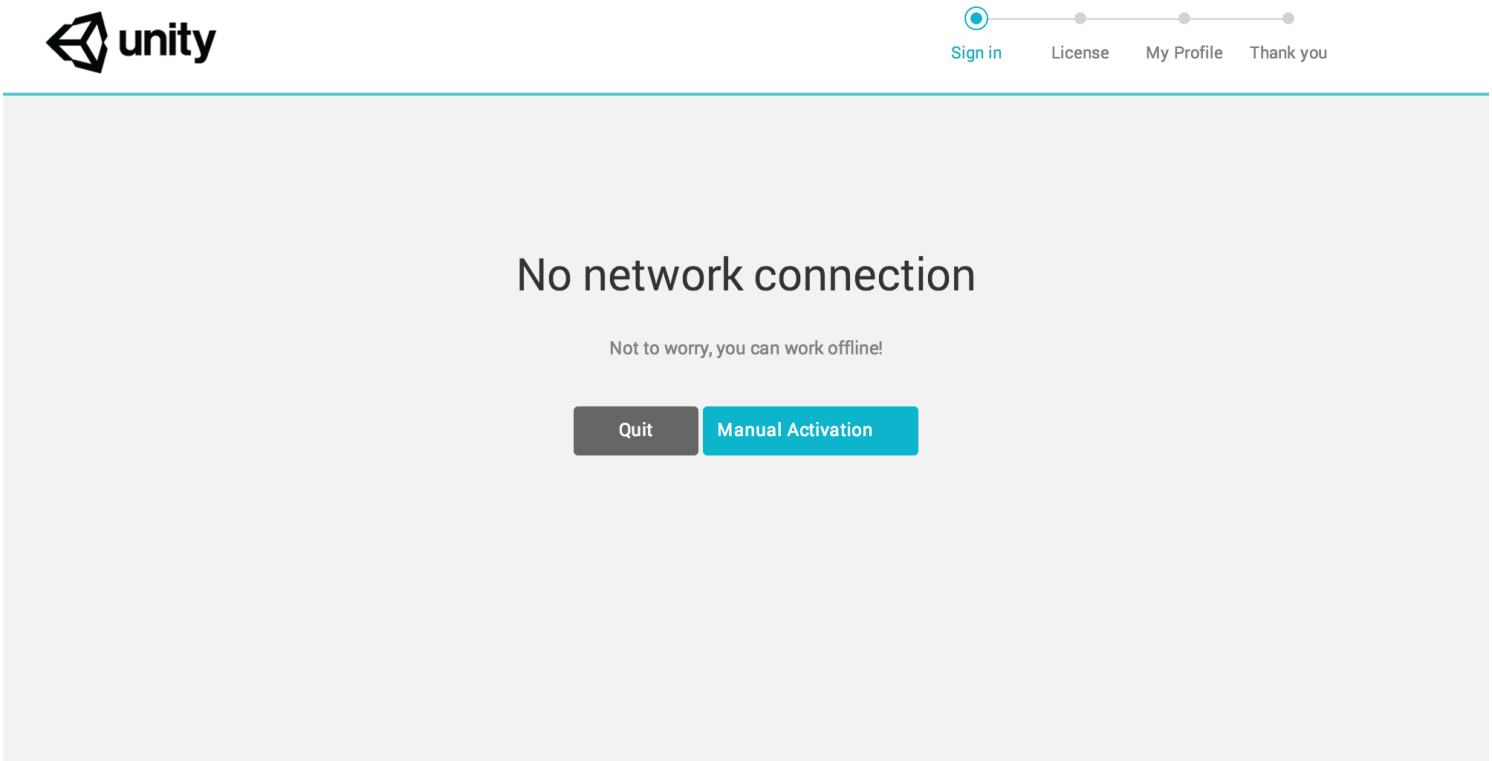
You do not have an Internet connection
You are behind a firewall, proxy or anti-viral software

If either of the above is true, Unity will not be able to contact the license server and the Editor will automatically attempt to perform a manual activation.

Follow the steps here to manually activate Unity on your machine:

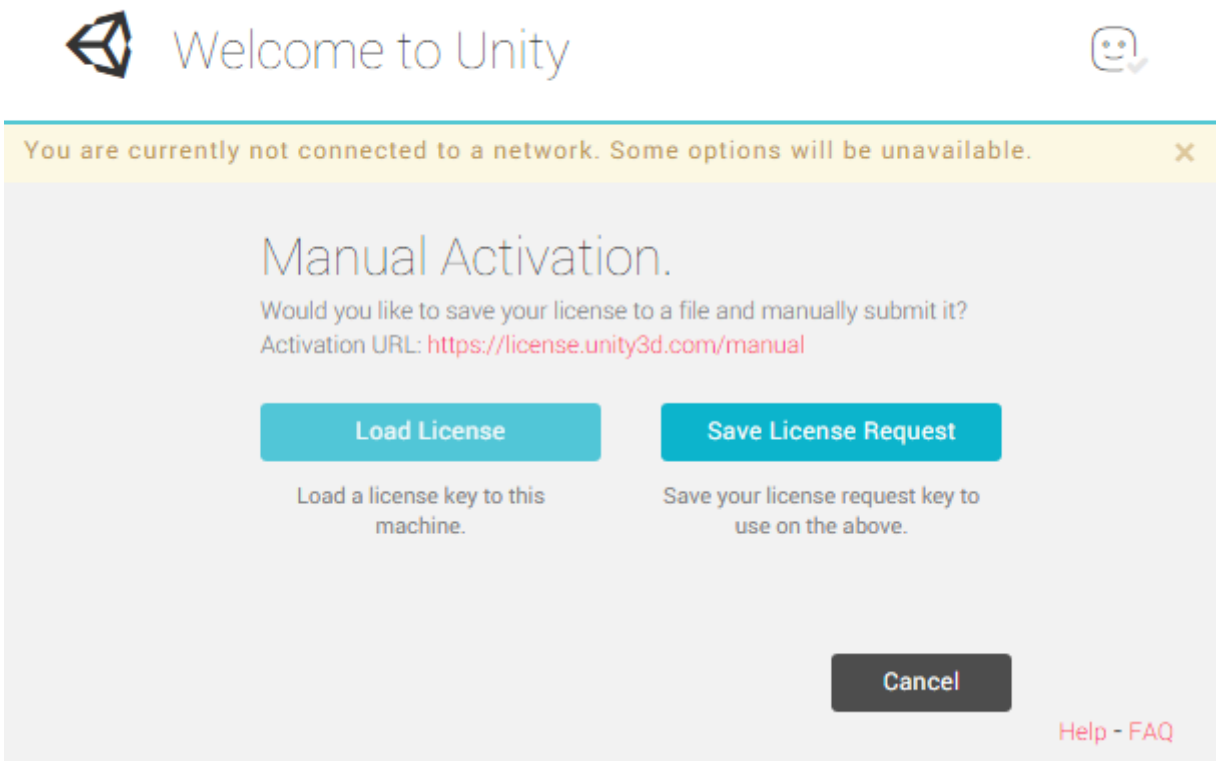
(Please note that you need to have access to a machine with Internet access as part of this process, but this does not have to be the machine on which you are trying to activate Unity.)

Open Unity. If Unity cannot contact the license server, you will be presented with the following window.

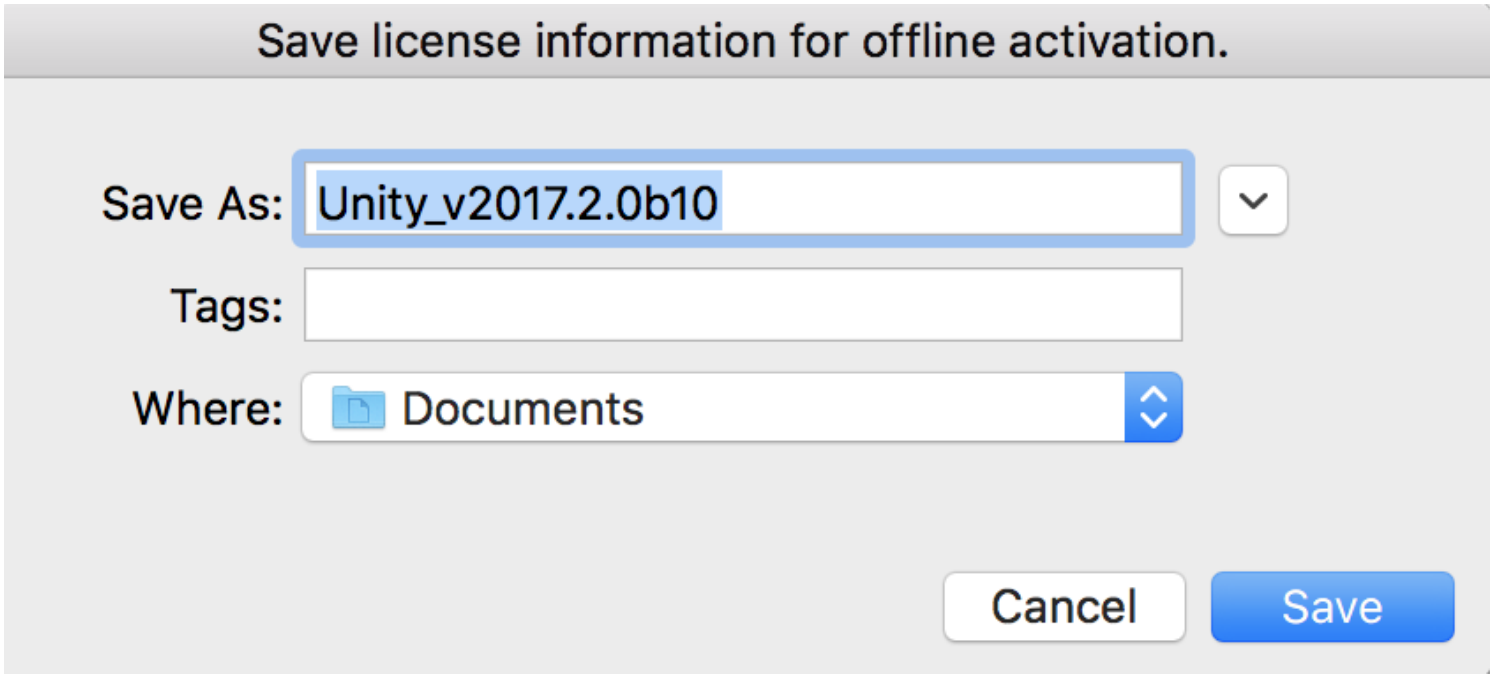


Click the **Manual Activation** button.

Click the **Save License Request** button to save the License Request file.



Save the license request file by selecting the **Save License Request** button. Save the file in a directory of your choice. (Make sure you remember where you save the file. In this example, the license file is being saved in the **Documents** folder).

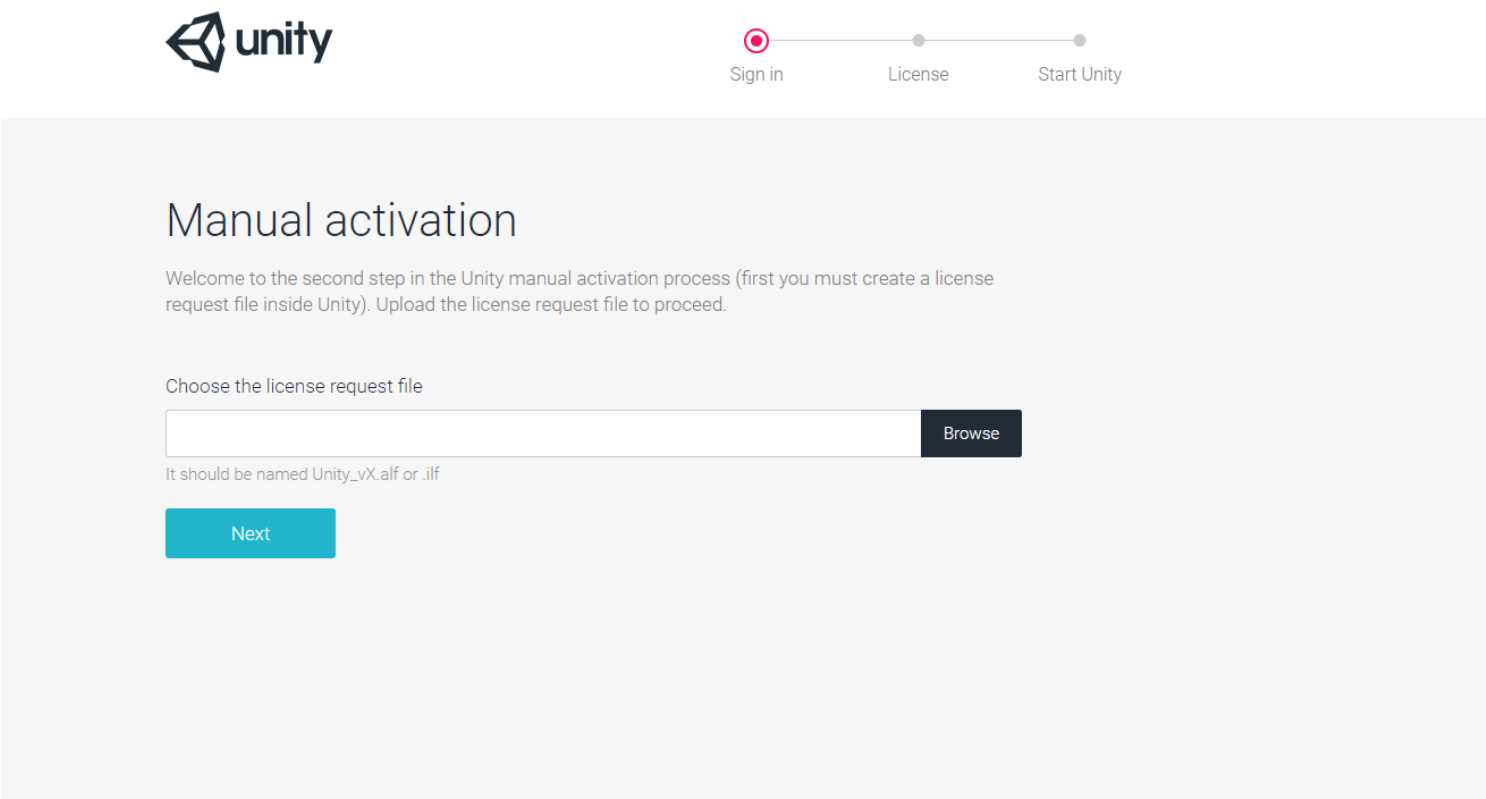


*Once you press the **Save** button, the file is saved and you should see the following notification at the top of the window.*

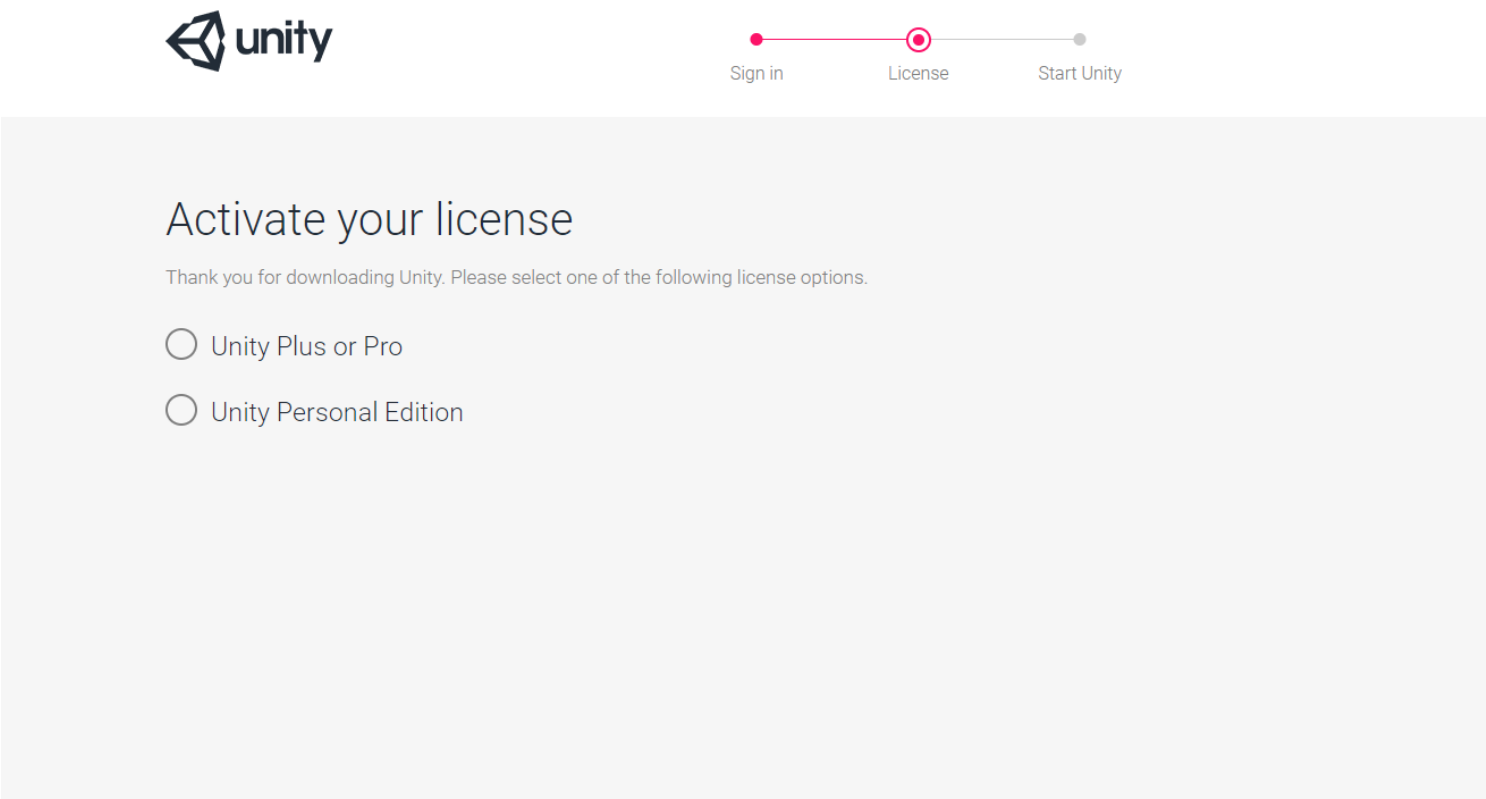
The license request file you just created is tied to the machine you generated it on. This license file will not work on any other machine. The license file will stop recognising a machine that has been reformatted or has had hardware changes.

The next steps require Internet access. If your machine does not have Internet access, you can activate your license on a machine that does have access by copying the file to the other machine, activating the license using the following steps, and then copying the file back to your machine to use Unity.

Go to the **Manual Activation** page. A new window will appear:

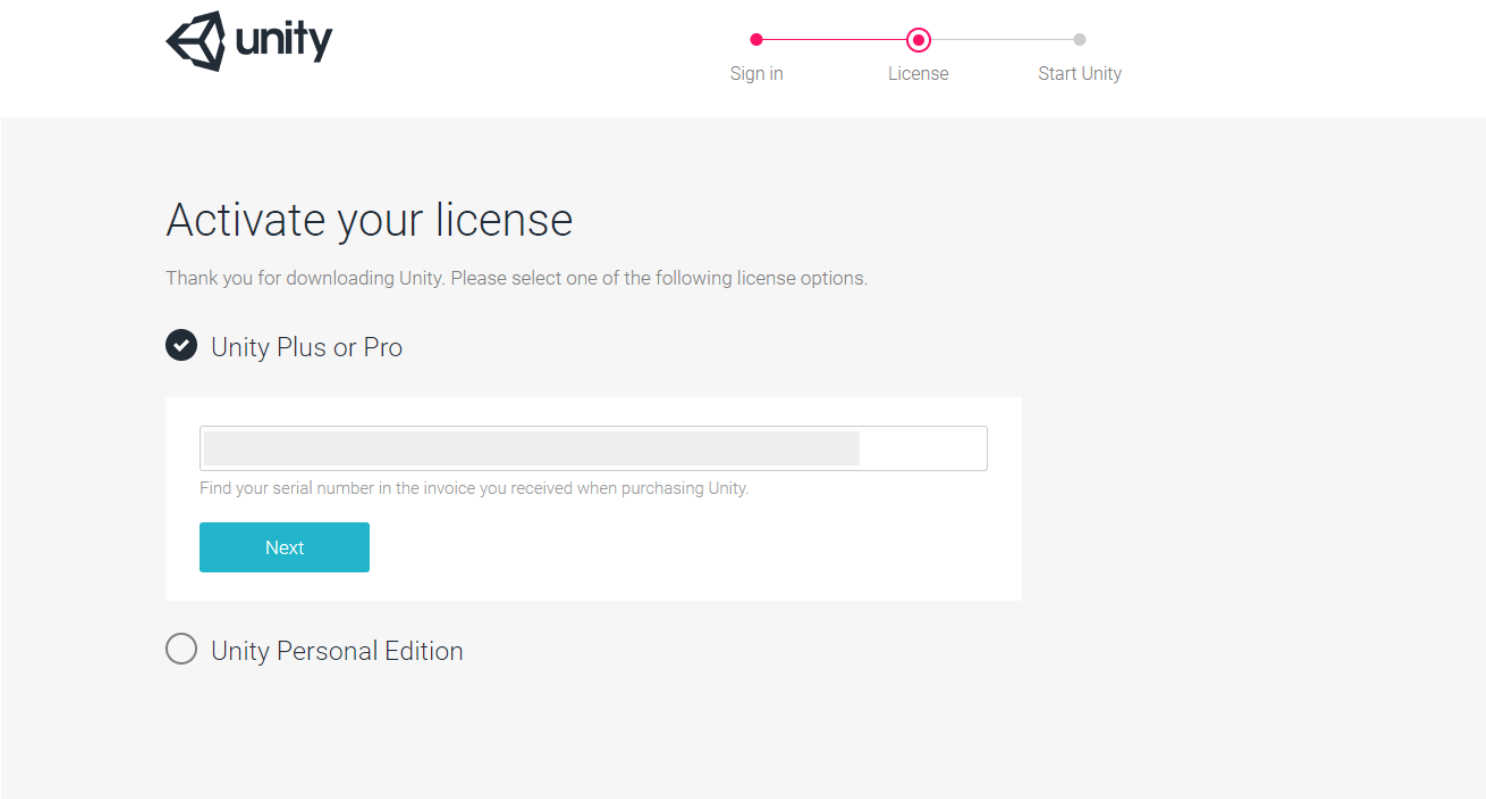


Click the **Browse** button to choose the file that you saved to your **Documents** folder, and then click the **Next** button. The following window will appear.



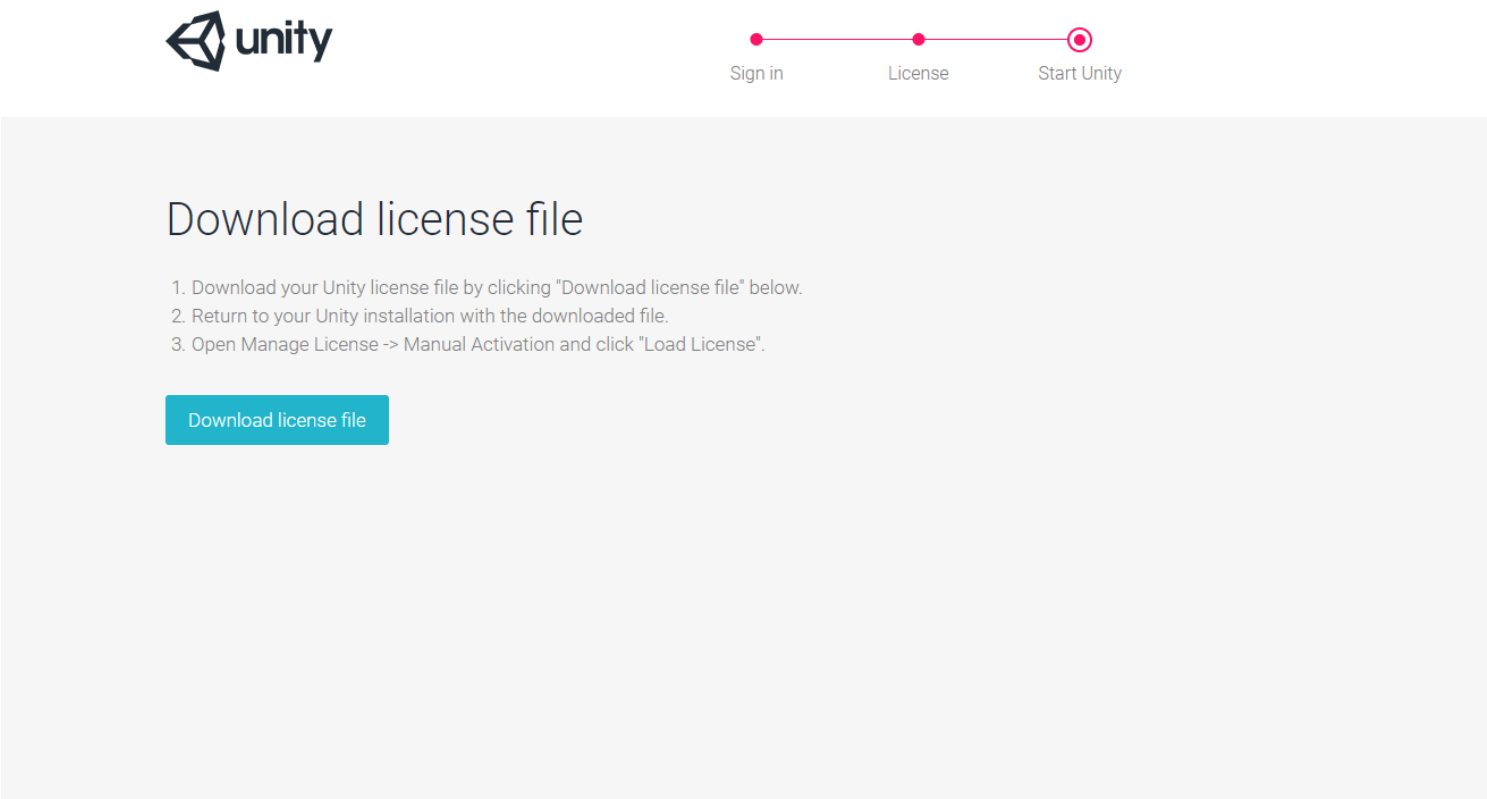
In this example, we are licensing a Plus or Pro version of Unity.

Enter the serial number.

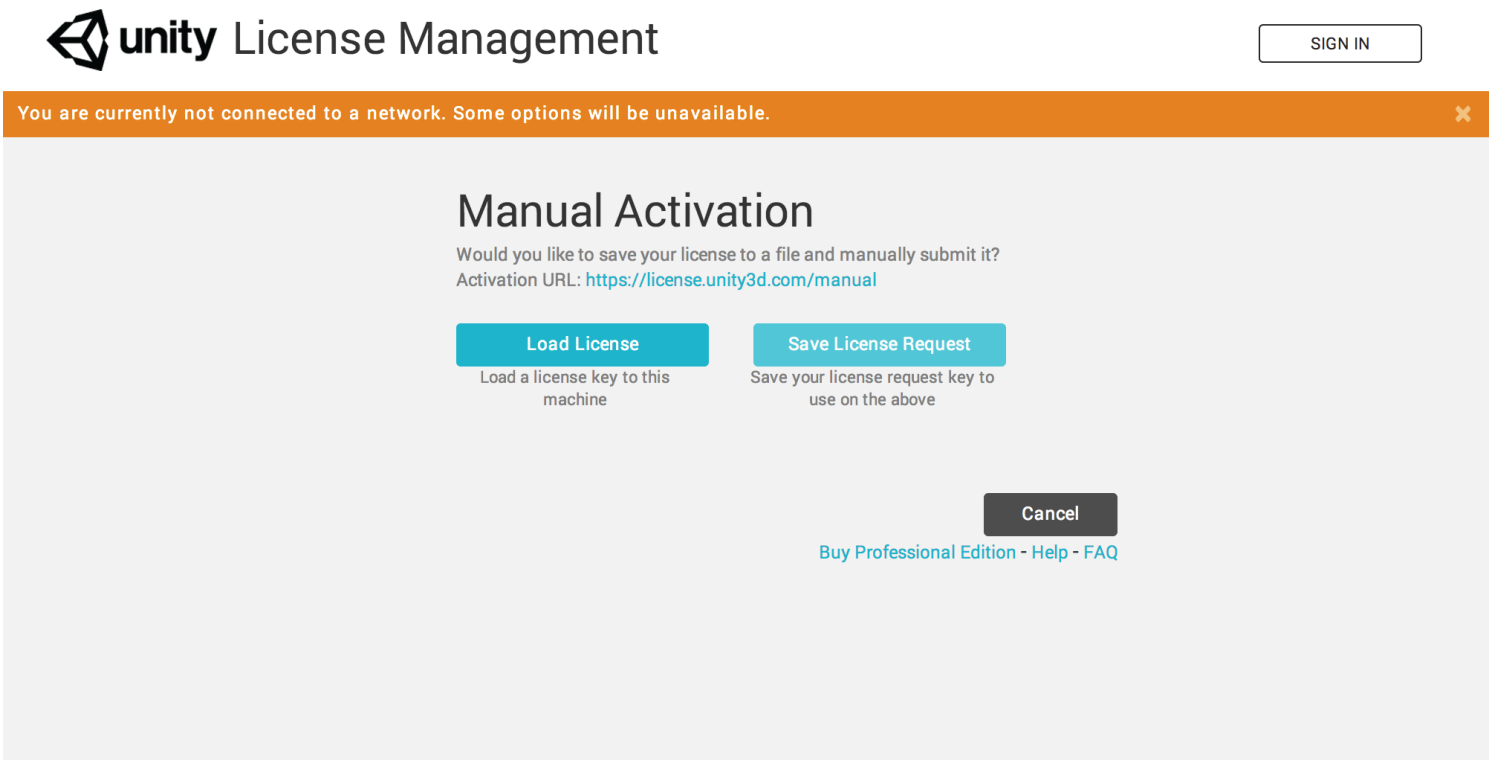


Note: In this window the serial number has been hidden. Click the **Next** button.

Click **Download license file**. This will save the license file to a location you select..



Go back to Unity. Click the **Load License** button to load the serial number file which was downloaded.



This will open up your directories within your hard drive.

Now, select the file that you just downloaded and click **OK**. A “Thank you!” window will appear. Press the **Start Using Unity** button to continue.

Note: Some browsers may append ".xml" to the license file name. If this is the case, you will need to delete this extension before attempting to load the license file.

If you have already activated Unity and, for example, need to update your license, see the Managing your Unity license page.

For any further assistance, please contact **support@unity3d.com**.

2017–09–06 Page amended with limited editorial review
License activation updated in Unity 2017.2

Managing your License

Leave feedback

If your machine is no longer accessible:

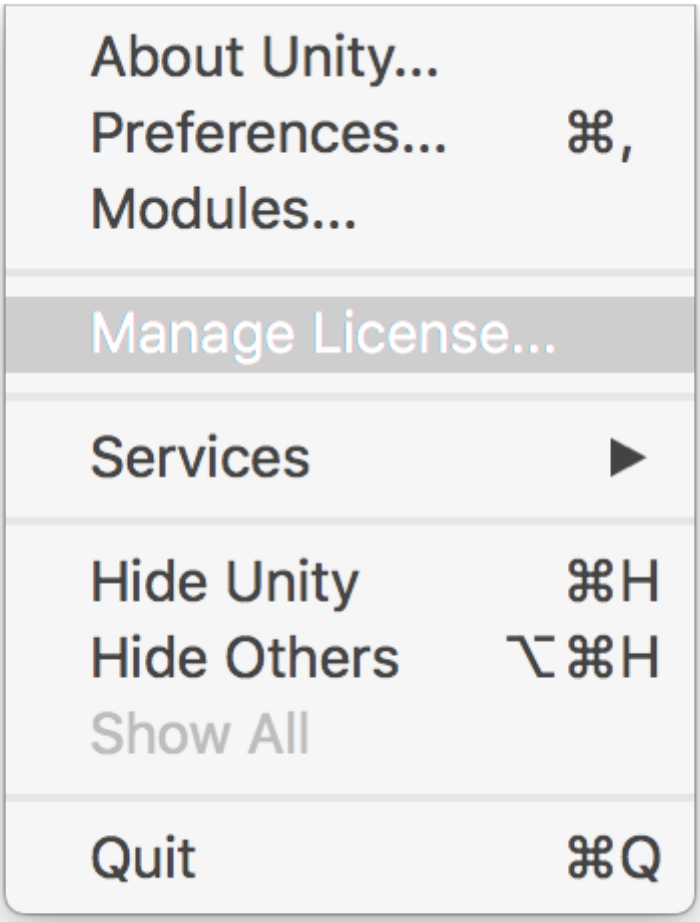
You can return activations from the serial number(s) added to your webstore account.

Go to the License Management Section of your webstore account.
Add your serial number to the account if it is not already added.
Click **Activations** to the right of the serial number.
Click **Disable all Activations**.
Unfortunately you cannot return single activations, this only resets them.

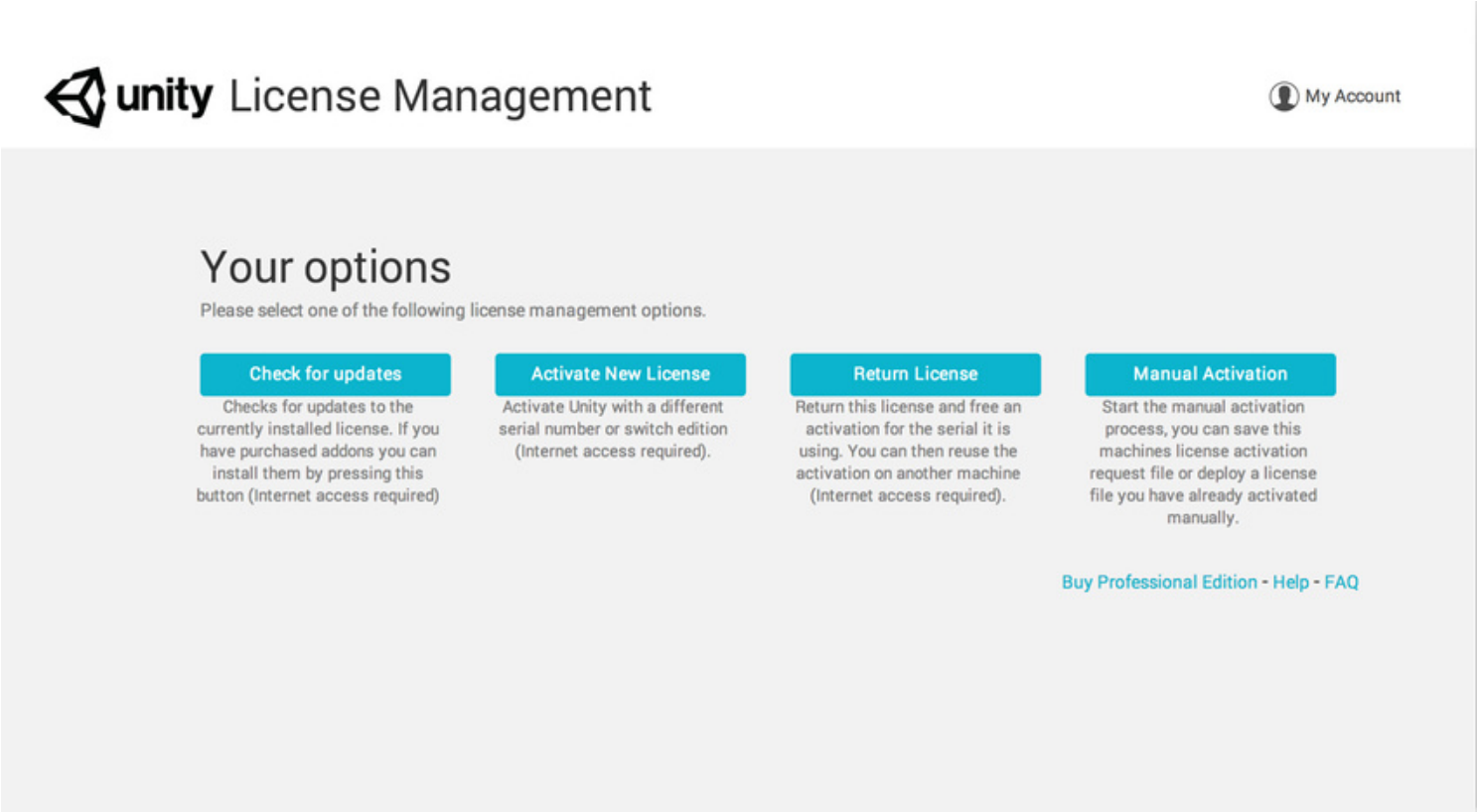
If your machine is still accessible:

You are now able to manage your license from the editor. Below is a guide to how this system works and performs.

Click the **Unity** drop-down on your **toolbar** (**Help** on Windows OS).
Click the **Manage License** option. (This is the unified place within the Editor for all your licensing needs).



The **License Management** window will appear. You then have four options, explained below:



License Management Window

Check for updates	Cross-references the server, querying your Serial number for any changes that may have been made since you last activated. This is handy for updating your license to include new add-ons once purchased and added to your existing license via the Unity Store.
Activate New License:	Enables you to activate a new serial number on the machine you're using.
Return License	Enables you to return the license on the machine in question, in return for a new activation that can be used on another machine. Once clicked, the Editor will close and you will be able to activate your serial number elsewhere. For more information on how many machines a single license enables use on, please see our EULA.
Manual Activation	Enables you to activate your copy of Unity offline. This is covered in more depth here.

If you encounter any error codes or other problems, please refer to the Unity Support Knowledge Base or contact Unity Customer Support for more assistance.

2017–09–06 Page amended with limited editorial review

License activation updated in Unity 2017.2

Activation FAQ

Leave feedback

How many machines can I install my copy of Unity on?

Every paid commercial Unity license allows a *single* person to use Unity on *two* machines that they have exclusive use of. Be it a Mac and a PC or your Home and Work machines. Educational licenses sold via Unity or any one of our resellers are only good for a single activation. The same goes for Trial licenses (Unity 4.x only), unless otherwise stated.

The free version of Unity may not be licensed by a commercial entity with annual gross revenues (based on fiscal year) in excess of US$100,000, or by an educational, non-profit or government entity with an annual budget of over US$100,000.

If you are a Legal Entity, you may not combine files developed with the free version of Unity with any files developed by you (or by any third party) through the use of Unity Pro. See our EULA for further information regarding license usage.

I need to use my license on another machine, but I get that message that my license has been 'Activated too many times'. What should I do?

You will need to 'Return' your license. This enables you to return the license on the machine you no longer require, which in turn enables you to reactivate on a new machine. Refer to Managing your Unity License, for more information.

My account credentials aren't recognised when logging in during the Activation process?

Ensure that your details are being entered correctly. Passwords ARE case sensitive, so check you are typing exactly as you registered. You can reset your password using this link.

If you are still having issues logging in, contact support@unity3d.com

Can I use the current version of Unity with a Serial number from a previous version?

No, you cannot. In order to use the current version of Unity, you need to upgrade your license. You can do this Online, via our Web Store

I am planning to replace an item of hardware and/or my OS. What should I do?

As with changing machine, you will need to 'Return' your license before making any hardware or OS changes to your machine. If you fail to 'Return' the license, our server will see a request from another machine and inform you that you have reached your activation limit for the license. Refer to the Managing your Unity License, for more information regarding the return of a license.

My machine died without me being able to 'Return' my license, what now?

First, try visiting:

https://store.unity3d.com/account/licenses

This page should allow you to return activations, through use of the "Disable all activations" button.

If the online page doesn't help you, then email support@unity3d.com explaining your situation. Include the details below.

The serial number you were using on the machine.
The (local network) name of the machine that died.
The order or invoice number used to make the purchase.
The Support Team will then be able to 'Return' your license manually. This can take some time. Note that this process is not possible for licenses that have not been purchased.

I have two licenses, each with an add-on I require, how do I activate them in unison on my machine?

You cannot. A single license may only be used on one machine at any one time.

Where is my Unity license file stored?

`/Library/Application Support/Unity` (OS X)
`C:\ProgramData\Unity\` (Windows)
`C:\Users\(username)\AppData\Local\VirtualStore\ProgramData\Unity`, if Windows User Account Control (UAC) has restricted your access to `C:\ProgramData\Unity`. (This can happen if the folder is deleted or Unity is started with administrative permissions first time.)

How can I use different Unity versions?

Unity assumes that only a single version of Unity can be run on a machine. You can, however, have multiple Unity versions installed and run on your machine. These versions will all need the same serial number. If you have different versions of Unity which need different licenses then you will need a way to copy licenses around. One way to do this is have the licenses stored on your Desktop. Before running a specific version copy the requiired ULF file into the location where the license needs to be stored.

For any further assistance, please contact at support@unity3d.com.

Upgrade Guides

Leave feedback

Visit the pages below for information about upgrading to later versions of Unity.

Upgrading to Unity 2018.3 [Public beta expected August 2018]

Upgrading to Unity 2018.2 [Released 10 July 2018] - [Public beta 07 May 2018]

Upgrading to Unity 2018.1 [Released 02 May 2018] - [Public beta 10 January 2018]

Upgrading to Unity 2017.3 [Released 19 December 2017] - [Public beta 25 September 2017]

Upgrading to Unity 2017.2 [Released 12 October 2017] - [Public beta 14 July 2017]

Upgrading to Unity 2017.1 [Released 12 July 2017] - [Public beta 11 April 2017]

Upgrading to Unity 5.6 [Released 31 March 2017] - [Public beta 12 December 2016]

Upgrading to Unity 5.5 [Released 30 November 2016] - [Public beta 29 August 2016]

Upgrading to Unity 5.4 [Released 28 July 2016] - [Public beta 14 March 2016]

Upgrading to Unity 5.3 [Released 08 December 2015]

Upgrading to Unity 5.2 [Released 08 September 2015]

Upgrading to Unity 5.0 [Released 03 March 2015]

Upgrading to Unity 4.0 [Released 13 November 2012]

Upgrading to Unity 3.5 [Released 14 February 2012]

Using the Automatic API Updater

Leave feedback

Sometimes, during development of the Unity software, we make the decision to change and improve the way the classes, functions and properties (the API) work. We do this with a focus on causing the least impact on user's existing game code, but sometimes in order to make things better, we have to break things.

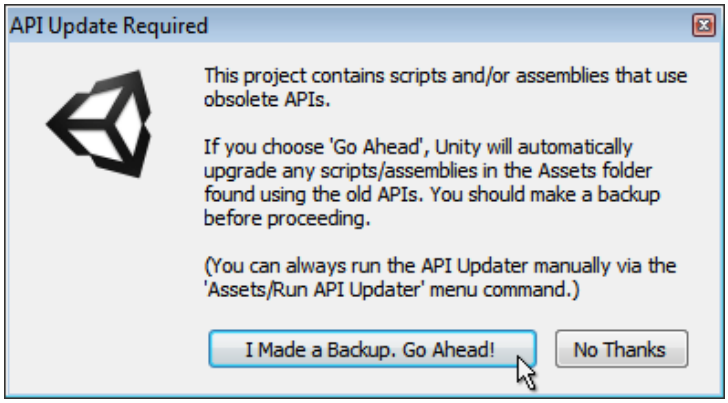
We tend to only introduce these significant "breaking changes" when moving from one significant version of Unity to another, and only in cases that it makes Unity easier to use (meaning users will incur fewer errors) or brings measurable performance gains, and only after careful alternative consideration. However, the upshot of this is that if you were to - for example - open a Unity 4 project in Unity 5, you might find some of the scripting commands that you used have now been changed, removed, or work a little differently.

One obvious example of this is that in Unity 5, we removed the "quick accessors" which allowed you to reference common **component** types on a **GameObject** directly, such as `gameObject.light`, `gameObject.camera`, `gameObject.audioSource`, **etc**.

In Unity 5, you now have to use the GetComponent command for all types, except transform. Therefore if you open a Unity 4 project that uses `gameObject.light` in Unity 5, you will find that particular line of code is *obsolete* and needs to be updated.

The automatic updater

Unity has an **Automatic Obsolete API Updater** which will detect uses of obsolete code in your **scripts**, and can offer to automatically update them. If you accept, it will rewrite your code using the updated version of the API.



The API Update dialog

Obviously, as always, it's important to have a backup of your work in case anything goes wrong, but particularly when you're allowing software to rewrite your code! Once you've ensured you have a backup, and clicked the "Go Ahead" button, Unity will rewrite any instances of obsolete code with the recommended updated version.

For example you had a script which did this:

```
light.color = Color.red;
```

Unity's API updater would convert that for you to:

```
GetComponent<Light>().color = Color.red;
```

The overall workflow of the updater is as follows:

Open a project / import a package that contains scripts / assemblies with obsoleted API usage

Unity triggers a script compilation

API updater checks for particular compiler errors that it knows are “updatable”

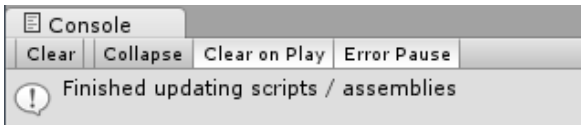
If any occurrence is found in the previous step, show a dialog to user offering automatic update, otherwise, we’ve finished.

If user accepts the update, then run API updater (which will update all scripts written in the same language being compiled in step 2)

Go to step 2 (to take any updated code into account) until no scripts get updated in step 5

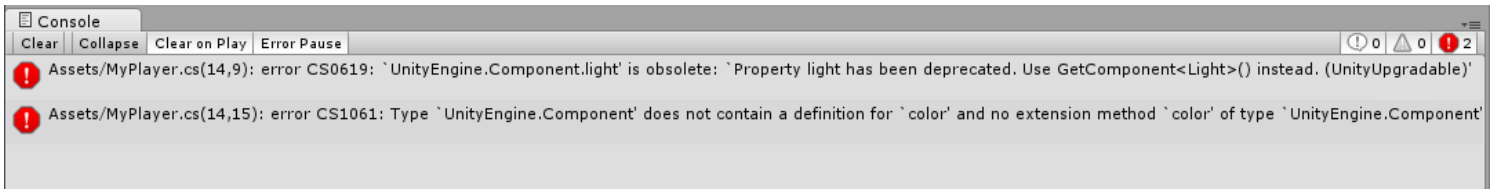
So, from the list above you can see the updater may run multiple times if there are scripts which fall into different compilation passes (Eg, scripts in different languages, editor scripts, etc) that use obsolete code.

When the API Updater successfully finishes, the console displays the following notification:



Success notification

If you choose *not* to allow the API updater to update your scripts, you will see the script errors in your console as normal. You will also notice that the errors which the API Updater could update automatically are marked as **(UnityUpgradable)** in the error message.



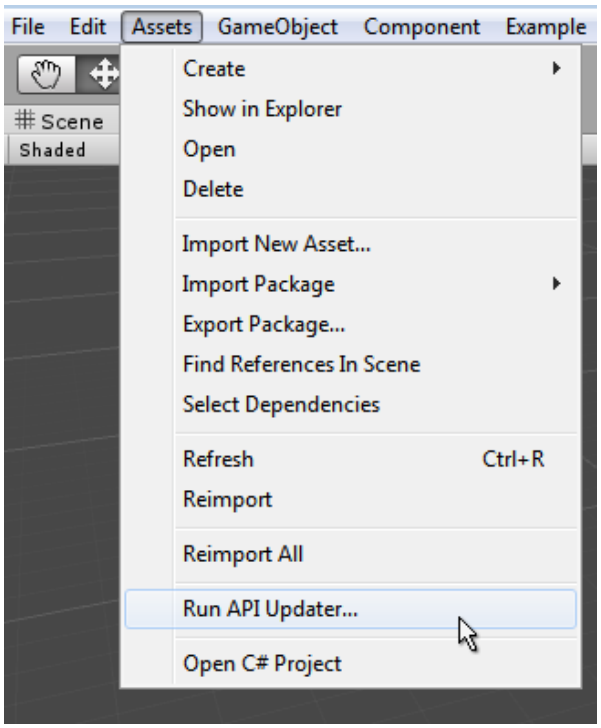
Errors in the console, when the API updater is canceled

If your script has other errors, in addition to obsolete API uses, the API updater may not be able to fully finish its work until you have fixed the other errors. In this case, you’ll be notified in the **console window** with a message like this:

Other errors in your scripts can prevent the API updater from working properly.

“Some scripts have compilation errors which may prevent obsolete API usages to get updated. Obsolete API updating will continue automatically after these errors get fixed.”

Once you have fixed the other errors in your script, you can run the API updater again. The API updater runs automatically when a script compilation is triggered, but you can also run it manually from the **Assets** menu, here:



The API Updater can be run manually from the Assets menu.

Command line mode

When running Unity in batch mode from the command line, use the `-accept-apiupdate` option to allow the API Updater to run. For more information, see Command Line Arguments.

Logging

A Version control system helps you see which changes the APIUpdater applies to a Project's scripts. However, this can be difficult when dealing with pre-compiled assemblies. To see the list of changes made by the AssemblyUpdater (the APIUpdater component responsible for updating assemblies), set the `UNITY_ASSEMBLYUPDATE_LOGTHRESHOLD` environment variable to the desirable log threshold and start Unity. For example, on Windows you can enter:

```
c:> set UNITY_ASSEMBLYUPDATE_LOGTHRESHOLD=Debug

c:> \path\to\unity\Unity.exe
```

After AssemblyUpdater has finished, you can see the updates that have been applied in the Editor.log:

```
[AssemblyUpdater] Property access to 'UnityEngine.Rigidbody
UnityEngine.GameObject::get_rigidbody()' in 'System.Void
Test.ClassReferencingObsoleteUnityAPIThroughEditorAssembly::Run()' replaced with 'T
UnityEngine.GameObject::GetComponent<UnityEngine.Rigidbody>()'.
```

The valid values for the `UNITY_ASSEMBLYUPDATE_LOGTHRESHOLD` environment variable are, in increasing order of detail:

Error: The AssemblyUpdater only logs **Error** messages. Error messages are logged when the AssemblyUpdater fails to apply a specific update, which requires you to take corrective action (usually requesting the original assembly author to provide an updated version of the assembly).

Warning: The AssemblyUpdater only logs **Warning**, and **Error** messages. Warning messages usually indicate that the AssemblyUpdater has reached a state that could be a potential problem. These problems can depend on conditions not known to the AssemblyUpdater at the point the message was logged.

Info: The AssemblyUpdater only logs **Informational**, **Warning**, and **Error** messages. Info messages include updates applied by the AssemblyUpdater.

Debug: The AssemblyUpdater logs all messages. Debug help with troubleshooting. You may want to set the threshold to this level if you are having issues with the AssemblyUpdater and you want to report them to Unity.

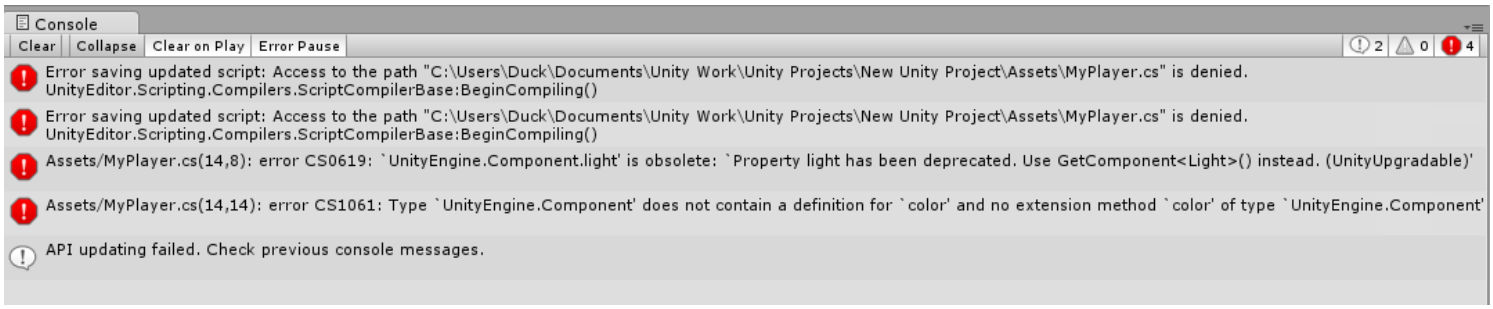
Error is the default value when `UNITY_ASSEMBLYUPDATE_LOGTHRESHOLD` is not set.

Troubleshooting

If you get a message saying "API Updating failed. Check previous console messages." this means the API updater encountered a problem that prevented it from finishing its work.

A common cause of this is if the updater was unable to save its changes - For example - the user may not have rights to modify the updated script because it is write protected.

By checking the previous lines in the console as instructed, you should be able to see the problems that occurred during the update process.



In this example the API updater failed because it did not have write permission for the script file.

Limitations

The API updater cannot automatically fix every API change. Generally, API that is upgradable is marked with `(UnityUpgradable)` in the obsolete message. For example:

```
[Obsolete("Foo property has been deprecated. Please use Bar (UnityUpgradable)")]
```

The API updater only handles APIs marked as `(UnityUpgradable)`.

The API updater might not run if the only updatable API in your scripts include component or GameObject common properties, and those scripts only access members of those properties. Examples of common properties are `renderer` and

`rigidbody`, and example members of those properties are `rigidbody.mass` and `renderer.bounds`. To workaround this, add a dummy method to any of your scripts to trigger the API updater. For example:

```
private object Dummy(GameObject o) { return o.rigidbody;}.
```

2018–07–31 Page amended with limited editorial review

“accept-apiupdate” command line option added in Unity 2017.2

AssemblyUpdater logging improved in Unity 2018.3

Upgrading to Unity 2018.3 beta

Leave feedback

2018.3 BETA DOCUMENTATION

This is first draft documentation for Unity 2018.3, which is is currently in beta. As such, the information in this document may be subject to change before final release. Note that this guide is for a beta release and may not list all known upgrade issues.

See first draft documentation on Upgrading to Unity 2018.3b.

For more information about what's new in 2018.3b and for more detailed upgrade notes, see the 2018.3 Beta Release Notes.

Upgrading to Unity 2018.2

Leave feedback

This page lists any changes in 2018.2 which might affect existing projects when you upgrade from earlier versions of Unity *

UIElements.ContextualMenu changes

The `UIElements.ContextualMenu` menu action callbacks now takes a `ContextualMenu.MenuAction` parameter instead of an `EventBase` parameter.

`ContextualMenu.InsertSeparator` takes an additional `string` parameter.

Multiplayer: The deprecated legacy networking APIs have been removed (RakNet based)

This feature was deprecated in Unity 5.1 and has now been removed. You can no longer use it in your projects in Unity 2018.2.

See the Multiplayer and Networking section for more information on networking in Unity.

Photoshop Data file (PSD) import using transparency

When having actual transparency, Photoshop will tweak pixel colors to blend them with matte (background) color. The process of preparing alpha channels properly is described in our how to prepare alpha channels documentation.

You can ignore dilation in this document, the important part is the note that states that you want to have an “opaque” image with a separate alpha channel/mask (instead of having transparency). Unity tweaked colors to “remove” matte, but this has ceased as of 2018.2. If you had PSD with transparency you might start seeing white color on the edges. To fix this, consult the manual link above and make an actual alpha channel (instead of transparency).

For more information about what’s new in 2018.2 and for more detailed upgrade notes, see the 2018.2 Release Notes. |

Upgrading to Unity 2018.1

Leave feedback

This page lists any changes in 2018.1 which might affect existing projects when you upgrade from earlier versions of Unity

Coroutines returned from a MonoBehaviour while its GameObject is being disabled or destroyed are no longer started.

Historically, when a **GameObject** is disabled or destroyed, it stops all running coroutines on its children MonoBehaviours. In certain cases, however, coroutines started from methods called during these times (for example, `OnBecameInvisible()`) were previously allowed to start. This led to **component** order-specific behavior and, in some cases, crashes.

In Unity 2018.1, coroutines returned during GameObject disable or destroy are no longer started.

BuildPipeline APIs now return a BuildReport object instead of a string

The BuildPipeline APIs, such as `BuildPipeline.BuildPlayer`, and `BuildPipeline.BuildAssetBundles`, previously returned a string. This was empty if the build succeeded and contained an error message if the build failed.

In 2018.1, this has been replaced with the new BuildReport object, which contains much richer information about the build process.

To check whether the build succeeded, retrieve the `summary` property of the report object, and check its *result* property - it will be `BuildResult.Succeeded` for a successful build. For example:

```
var report = BuildPipeline.BuildPlayer(...);

if (report.summary.result != BuildResult.Succeeded)
{
    throw new Exception("Build failed");
}
```

Player Quit notifications have changed from messages to Events

Previously, to be notified when the Unity standalone player was quitting, you would implement the `OnApplicationQuit` method on a MonoBehaviour and to abort the player from quitting you would call `Application.CancelQuit`.

Two new events have been introduced. These are `Application.wantsToQuit` and `Application.quitting`. You can listen to these events to get notified when the Unity standalone player is quitting. `Application.wantsToQuit` is called when the player is intending to quit, the listener for `wantsToQuit` must

return true or false. Return true if you want the player to continue quitting or false to abort the quit. The `Application.quitting` event is called when the player is guaranteed to quit and cannot be aborted.

`Application.CancelQuit` has been deprecated, please use the `Application.wantsToQuit` instead.

```
using UnityEngine;

public class PlayerQuitExample

{

    static bool WantsToQuit()

    {

        // Do you want the editor to quit?

        return true;

    }

    static void Quit()

    {

        Debug.Log("Quitting the Player");

    }

    [RuntimeInitializeOnLoadMethod]

    static void RunOnStart()

    {

        Application.wantsToQuit += WantsToQuit;

        Application.quit += Quit;

    }

}
```

Deprecating AvatarBuilder.BuildHumanAvatar on .Net platform

This change affect the following runtime platform: WSAPlayerX86, WSAPlayerX64, and WSAPlayerARM.

There is no replacement for now.

TouchScreenKeyboard.wasCanceled and TouchScreenKeyboard.done have been made obsolete

There is a new TouchScreenKeyboard.status that can be queried to cover the deprecated states and more states.

MonoDevelop 5.9.6 removed from Unity Installers and support for it has been deprecated in Unity.

MonoDevelop 5.9.6 has been replaced by Visual Studio for Mac on macOS as the bundle C# script editor in the macOS installer. Visual Studio 2017 Community is now the only C# script editor installed with Unity on Windows.

When it is installed in the default location next to the Unity executable, Unity no longer recognises MonoDevelop as the “MonoDevelop (built-in)” external script editor in preferences. When no C# code editor is installed and selected in preferences, Unity uses the system default application for opening C# (.cs) scripts.

BuildPipeline callback interfaces now take a BuildReport object

The BuildPipeline callback interfaces: `IPreprocessBuild`, `IPostprocessBuild` and `IProcessScene` have been changed so that they now require you to pass in a `BuildReport` object. This replaces the previous parameters for build path / target platform; you will need to change your code if you are implementing these interfaces.

Both the build path and the target platform can be accessed via the `BuildReport` object. The build path is now `report.summary.outputPath` and the target platform is `report.summary.platform`.

Assets located in plugin folders will no longer be imported via specialized importers

Previously, assets located in plugin folders (for example, in directories with the extension .bundle, .plugin, or .folder) were imported using specialized importers. Textures were imported via texture importers, AudioClips via the audio importer, etc. Now all those assets will be imported using default importer, that means you won’t be able to reference those assets like you did before, because they no longer have a specialized type (Texture, AudioClip, etc). Plugin folders are contained packages, thus assets inside shouldn’t be accessible externally, except through plugin accessing techniques.

To continue using those assets, you’ll need to move them outside of your plugin folders.

Particle System Mesh particles applied the Pivot Offset value incorrectly

The mathematical formula used for applying pivot offsets to Meshes was incorrect, and was inconsistent with how it worked for Billboard particles. To achieve the correct scale, the Pivot Offset should be multiplied by the size of the particle, so a Pivot Offset of 1 is equal to one full width of the particle.

For Meshes, the size was being multiplied twice, meaning the pivot amount was based on the squared particle size. This made it impossible to get consistent results in systems containing varying sized particles.

For systems using particles of equal size, the formula can be reverse-engineered to decide how much to adjust the pivot offset by, to compensate for this change in behavior:

Old formula: **offset = size * size * pivot**

New formula: **offset = size * pivot**

Therefore, if all particles are of equal size:

newOffset = pivot / size

In systems where the size varies between particles, a visual reassessment of the systems in question will be necessary.

GPU Instancing supports Global Illumination

From 2018.1, Global Illumination (GI) is supported by GPU Instancing rendering in Unity. Each GPU instance can support GI coming from either different Light Probes, one lightmap (but different region in the atlas), or one Light Probe Proxy Volume component (baked for the space volume containing all the instances). Standard shaders and surface shaders come with these changes automatically, but you need to update custom shader code to enable these features.

Handle Draw and Size Function Defaults

Complex handles in the UnityEditor.IMGUI.Controls namespace, such as BoxBoundsHandle, CapsuleBoundsHandle, SphereBoundsHandle, ArcHandle, and JointAngularLimitHandle have delegates that can be assigned to, in order to alter the appearance of their control points. Previously, assigning a value of null to these delegates would fall back to a default behavior. Assigning a value of null to them now results in no behavior, making it easier to disable particular control handles. Respectively, each class now has public API points for the default methods if you need to reset the control handles to their default behavior.

Compiling ‘unsafe’ C# code in the Unity Editor now requires enabling of option.

Compiling ‘unsafe’ C# code now requires the **Allow ‘unsafe’ code** option to be enabled in the Player Settings for predefined assemblies (like Assembly-CSharp.dll) and in the inspector for Assembly Definition Files assemblies. Enabling this option will make Unity pass /unsafe option to the C# compiler when compiling scripts.

‘UnityPackageManager’ directory renamed to ‘Packages’

In 2017.2 and 2017.3, the Unity Package Manager introduced the **UnityPackageManager** directory, which was used to store a file named **manifest.json**. Package content could be accessed by scripts using virtual relative paths starting with **Packages**.

In 2018.1, the **UnityPackageManager** directory has been renamed to **Packages** for consistency with the virtual relative paths of packaged assets. The **manifest.json** file should automatically be moved to the new directory.

As a result:

If your project uses a Version Control System (VCS) such as Perforce or Git, it may be necessary to update its configuration to track the **Packages** directory instead of the **UnityPackageManager** directory.

If your project uses Nuget (or any external package manager) in a way that makes it use the **Packages** directory, its configuration should be changed to use a different directory. This is recommended to eliminate the small chance that a package be picked up by the Unity Package Manager, which could result in hard-to-debug issues like compilation errors or import errors.

To configure Nuget to use a different directory to store its packages, please refer to the official Microsoft documentation.
After migrating to the new directory, the **UnityPackageManager** directory can safely be deleted.

For more information about what's new in 2018.1 and for more detailed upgrade notes, see the 2018.1 Release Notes

Upgrading to Unity 2017.3

Leave feedback

This page lists any changes in 2017.3 which might affect existing projects when you upgrade from earlier versions of Unity.

For example:

Changes in data format which may require re-baking.

Changes to the meaning or behavior of any existing functions, parameters or **component** values.

Deprecation of any function or feature. (Alternatives are suggested.)

Lightmap intensity & emissive Materials in Enlighten

A bug introduced in 2017.2 increased the intensity of lightmaps generated for **Scenes** with static Meshes that have emissive Materials. This was the same for both baked and real-time emissive Materials. This has now been fixed in 2017.3 so the intensity is now similar to how it was in 2017.1.

This change affects any projects you have built in 2017.2 and then upgrade to 2017.3 or newer.

PassType.VertexLMRGBM has been deprecated

In Unity 2017.3, the **shader** pass **VertexLMRGB** is ignored. For example: `Tags { "LightMode" = "VertexLMRGBM" }`

Instead, provide or update a **VertexLM** shader pass using the DecodeLightmap shader function that supports all types of lightmap encodings. Built-in mobile shaders also use DecodeLightmap shader function now.

Lighting output may change in your existing projects that use built-in mobile shaders such as Mobile or VertexLit on desktop platforms. This is because the maximum range of RGBM encoded values has changed from [0, 8] to [0, 5].

2017–12–04 Page published with limited editorial review

Upgrading to Unity 2017.2

Leave feedback

This page lists any changes in 2017.2 which might affect existing projects when you upgrade from earlier versions of Unity.

For example:

Changes in data format which may require re-baking.

Changes to the meaning or behavior of any existing functions, parameters or **component** values.

Deprecation of any function or feature. (Alternatives are suggested.)

MonoBehaviour.OnValidate is now called when MonoBehaviour is added to a GameObject in the Editor

MonoBehaviour.OnValidate is called when a **Scene** loads, when GameObjects are duplicated or when a value changes in the **Inspector**. It is now also called when adding a MonoBehaviour to a GameObject in the Editor.

Scripting: InitializeOnLoad callback now invoked after deserialization

The callback timing for InitializeOnLoad has changed. It was previously invoked at a point that could lead to invalid object states for existing serialized objects when calling Unity API. It is now invoked after deserialization and after all objects have been created. As part of the creation of objects, the default constructor must be invoked. This change means that objects constructors are now invoked before InitializeOnLoad static constructors, whereas InitializeOnLoad was previously called before some object constructors.

Example:

```
[System.Serializable]
public class SomeClass
{
    public SomeClass()
    {
        Debug.Log("SomeClass constructor");
    }
}

public class SomeMonoBehaviour : MonoBehaviour
{
    public SomeClass SomeClass;
}

[InitializeOnLoad]
public class SomeStaticClass
{
    static SomeStaticClass()
    {
        Debug.Log("SomeStaticClass static constructor");
```

```
        }
    }
```

This would previously result in:
`SomeStaticClass static constructor` (InitializeOnLoad)
`SomeClass constructor` (object constructor)

After this change it will now be:
`SomeClass constructor` (object constructor)
`SomeStaticClass static constructor` (InitializeOnLoad)

New normal map type that support BC5 format.

Up to now Unity was supporting either RGB normal map or swizzled AG normal map (with x in alpha channel and y in green channel) with different **compression** format. There is now support for RG normal map (with x in red channel and y in green channel). UnpackNormal **shader** function have been upgraded to allow to use RGB, AG and RG normal map without adding **shader** variants. To be able to do this, the UnpackNormal function rely on having unused channel of the normal map set to 1. I.e a swizzled AG normal map must be encoded as (1, y, 1, x) and a RG (x, y, 0, 1). Unity normal map encoder enforce it.

There is no upgrade to do if users were using unmodified Unity. However in case users have done their own normal map shaders or their own encoding, they may need to take into account the need for swizzled AG normal map to be encoded as (1, y, 1, x). In case users were mixing normal map in swizzled AG before unpacking normal map, they may require to use UnpackNormalDXT5nm instead of UnpackNormal.

Always precompiled managed assemblies (.dlls) and assembly definition file assemblies on startup in the Editor.

Load precompiled managed assemblies (.dlls) and assembly definition file assemblies on Editor startup even if there are compile errors in other **scripts**. This is useful for Editor extension assemblies that should always be loaded on startup, regardless of other script compile errors in the project.

HDR emission.

If you are using precomputed realtime GI or baked GI, intense emissive materials set up in earlier versions of Unity could look more intense now, because their range is not capped any more. The RGBM encoding used previously gave an effective range of 97 for gamma space and 8 for linear color space. The HDR color picker had a maximum range of 99 so some materials could be set to be more intense than they seemed. After the upgrade, emission color is passed to the GI systems as true HDR 16 bit floating point values (range is now 64K). Internally, the realtime GI system is using the rgb9e5 shared exponent format that can represent these intense values but the baked **lightmaps** are limited by their RGBM encoding. HDR for baked **lightmaps** will be added in a later release.

VR to XR rename.

The UnityEngine.VR.* namespaces have been renamed to UnityEngine.XR.*. All types with VR in their name have also been renamed to their XR versions. For example: UnityEngine.VR.VRSettings is now UnityEngine.XR.XRSettings, **etc**.

The API updater has been configured to automatically update existing scripts and assemblies to the new type names and namespaces. If don't want to use the API updater, you can also manually update namespaces and types.

Namespace changes:

UnityEngine.VR -> UnityEngine.XR
UnityEngine.VR.WSA -> UnityEngine.XR.WSA
UnityEngine.VR.WSA.Input -> UnityEngine.XR.WSA.Input
UnityEngine.VR.WSA.Persistence -> UnityEngine.XR.WSA.Persistence
UnityEngine.VR.WSA.Sharing -> UnityEngine.XR.WSA.Sharing
UnityEngine.VR.WSA.WebCam -> UnityEngine.XR.WSA.WebCam
UnityEngine.VR type changes:

VRDevice -> XRDevice
VRNodeState -> XRNodeState
VRSettings -> XRSettings
VRStats -> XRStats
VRNode -> XRNode
All `VR.*` **profiler** entries have also been changed to `XR.*`.

UnityEngine.dll is now split into separate dlls for each UnityEngine module.

The UnityEngine.dll (which contains all public scripting API) has been separated into modules of code covering different subsystems of the engine. This makes the Unity code base better organized with cleaner internal dependencies, better for internal tooling and makes the code base more strippable. The separated modules include UnityEngine.**Collider** which is now in UnityEngine.PhysicsModule.dll and UnityEngine.Font which is now in UnityEngine.TextRendering.dll.

This change should typically not affect any of your existing projects and your scripts now automatically compiles against the correct assemblies. Unity now includes a UnityEngine.dll assembly file containing type forwarders of all UnityEngine types for all pre-compiled assemblies referencing the DLL which ensures backwards compatibility by forwarding the files to their new locations.

However, there is one case where existing code might break from this change. That is if your code uses reflection to get UnityEngine types, and assumes that all types live in the same assembly. This means such code would fail, because Collider and Font are now in different assemblies:

```
System.Type colliderType = typeof(Collider);
System.Type fontType = colliderType.Assembly.GetType("Font");
```

Getting either Collider or Font types from the “UnityEngine” assembly still works due to the use of type forwarders, like this:

```
System.Type.GetType("UnityEngine.Collider, UnityEngine")
```

Unity does still bundle a fully monolithic UnityEngine.dll which contains all UnityEngine APIs in the Unity Editor’s Managed/UnityEngine.dll folder. This makes sure that any existing Visual Studio/**MonoDevelop** solutions referencing UnityEngine.dll continues to build without needing to be updated to reference the new modular assemblies. You should continue to use this assembly to reference UnityEngine API in your custom solutions, as the internal split of modules is subject to change.

Material smoothness in Standard shader.

Purely smooth materials that use the GGX version of the standard shader now receive specular highlights which increases the realism of such materials.

2017–10–06 Page published with no editorial review

Upgrading to Unity 2017.1

Leave feedback

This page lists any changes in 2017.1 which might affect existing projects when you upgrade from earlier versions of Unity.

For example:

Changes in data format which may require re-baking.

Changes to the meaning or behavior of any existing functions, parameters or **component** values.

Deprecation of any function or feature. (Alternatives are suggested.)

UnityWebRequestTexture.GetTexture() nonReadable parameter change

This convenience API had a bug where nonReadable worked the opposite way then it should, that is setting it to true would result in a readable texture and vice versa. Now this has been corrected and parameter works the way it's documented. Note, that if you create DownloadHandlerTexture directly, you are not affected.

Particle System Stretched Billboard Pivot parameter change

X axis pivots are now accurate on Stretched Billboards. You may need to reconfigure your pivot settings on affected projects.

Y axis pivots are also accurate now and Unity automatically reconfigures these pivots when you upgrade your project.

Shader macro UNITY_APPLY_DITHER_CROSSFADE change

The macro was used for implementing the screendoor dithering effect for **LOD** cross-fading objects in the **fragment shader**. Previously you would need to pass the entire fragment IN structure to it. Now you only need to pass the screen position vector to it.

Remember that since 2017.1 the new `dithercrossfade` option is added for `#pragma surface` directive which can automatically generate the dithering code.

---

* 2018–04–12 Page amended with no editorial review

Upgrading to Unity 5.6

Leave feedback

This page lists any changes in 5.6 which might affect existing projects when you upgrade from earlier versions of Unity.

For example:

Changes in data format which may require re-baking.

Changes to the meaning or behavior of any existing functions, parameters or **component** values.

Deprecation of any function or feature. (Alternatives are suggested.)

Script serialization errors no longer work

The script serialization errors introduced in Unity 5.4 and described in detail in this blog post, will always throw a managed exception from 5.6 onwards.

Behaviour in Unity 5.5 is the same as in Unity 5.4.

PlayerSettings.apiCompatibilityLevel deprecated

This is now a per-platform setting. Use PlayerSettings.SetApiCompatibilityLevel and PlayerSettings.GetApiCompatibilityLevel instead. PlayerSettings.apiCompatibilityLevel will continue to function, but it will only affect the current active platform.

Lighting changes

Directional specular **lightmap** has been removed.

As a consequence, LightmapsMode.SeparateDirectional has been removed. Use LightmapsMode.CombinedDirectional instead.

Here are the prefered ways to get specular highlights in Unity 5.6:

For direct specular, stationary lights with real-time direct lighting provide high quality specular highlights in all modes but the subtractive one. Please see the *Mixed Lighting* documentation in the online lighting section of the 5.6 User Manual.

For indirect specular, use Reflection Probes, Screen Space Reflection (SSR), or both.

Mixed mode lighting has evolved

Mixed mode lighting in Unity 5.5 has been replaced with stationary lighting modes in Unity 5.6. This implies a lot of changes, and we advise you to carefully read the *Lighting Modes* documentation, and in particular the *Stationary Modes* draft documentation in the online lighting section of the 5.6 User Manual which details the newly available options.

Projects from pre-Unity 5.6 will be upgraded to **Subtractive** mode (see the *Subtractive Light Mode* in the online lighting section of the 5.6 User Manual. This is the closest match to the **Mixed** lighting from Unity 5.5.

However this mode is the lowest quality mode for Unity 5.6, so try other modes to see if they are a better fit for your projects. We highly recommend the **Shadowmask** (see the *Shadowmask* page and the *Distance Shadowmask* page in the online lighting section of the 5.6 User Manual, especially if you were using the now-defunct Directional Specular lightmaps before).

The default for new Unity 5.6 projects is **Distance shadowmask** mode. (See the *Distance shadowmask* page in the online lighting section of the 5.6 User Manual).

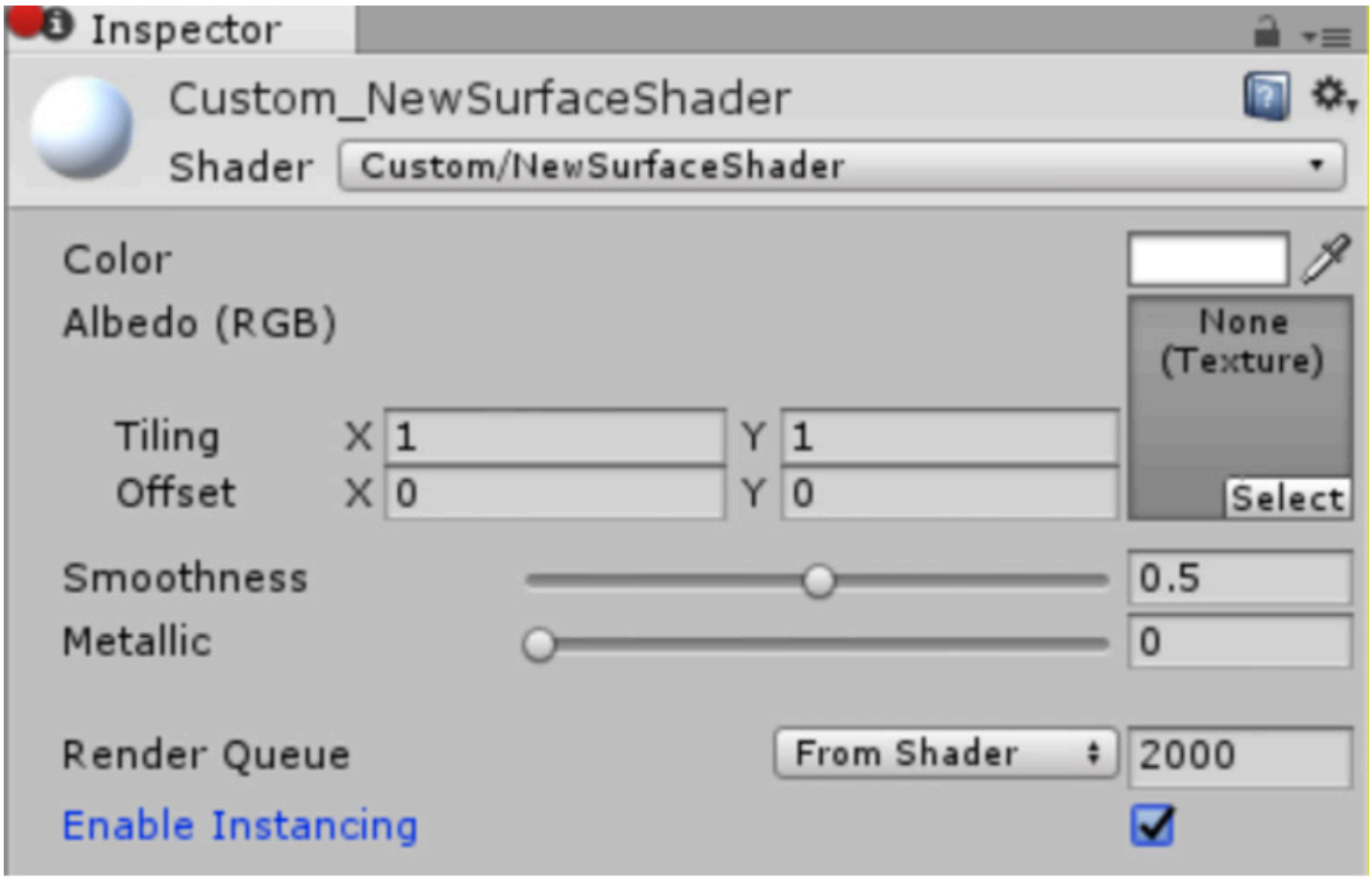
Light probes

In Unity 5.6 you can no longer choose whether direct lighting is added to Light Probes from the lighting window. This now happens automatically based on the lights types and the stationary modes. This ensures that direct lighting is not missing and that there is no double lighting on dynamic objects using Light Probes, greatly simplifying the process.

New GPU instancing workflow

You must now check the new **Enable Instancing** checkbox for all Materials used for GPU instancing. This is required for GPU instancing to work correctly.

Note that the checkbox is not checked automatically as it's impossible for Unity to determine if a Material uses a pre–5.6 **Shader**.



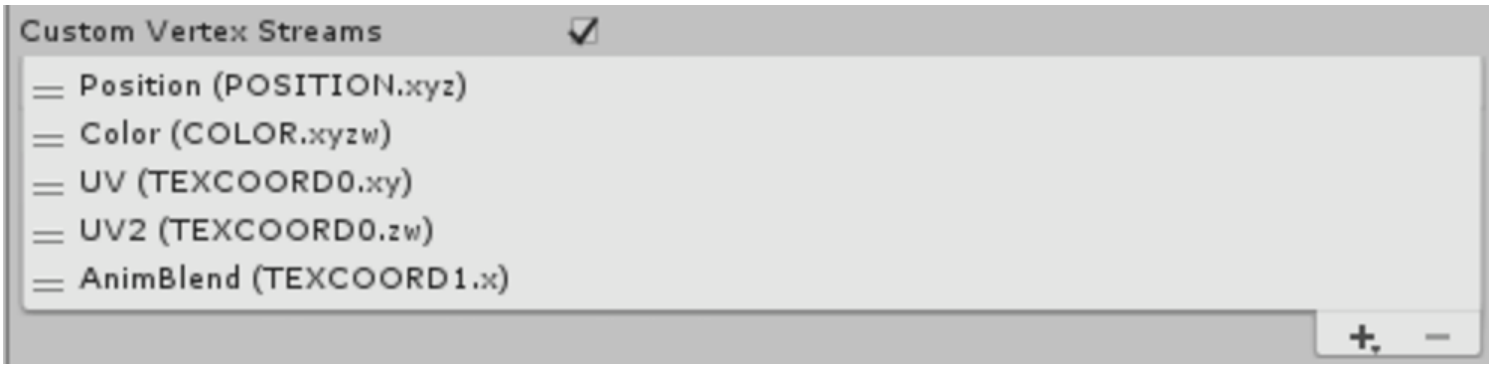
An upgrade error is also imposed on SpeedTree **Assets** to help you regenerate the SpeedTree Materials, so that you can have **Enable Instancing** checked.

Note: The newly introduced procedural instancing Shaders (those with `#pragma instancing_options procedural:func`) don't require this change because Shaders with the `PROCEDURAL_INSTANCING_ON` keyword are not affected.

Particle system changes

`Custom Vertex Streams` in the Renderer Module may now require you to manually upgrade your Particle Systems, if you use the `Particles/Alpha Anim Blend` Shader. In this situation, it is sufficient to simply remove the duplicated UV2 stream.

This is required because, in some cases, the duplicated stream is needed to maintain backwards compatibility, so there is no fully reliable auto-upgrade solution. The Normal and AnimFrame streams are also not required, but causes no problems if they exist. The fixed Particle System setup should look like this:



Secondly, the upgraded Emission Module will cause Animation Bindings attached the Burst Emission to be lost. It will be necessary to rebind those properties.

Animator change

Animate Physics: Rigidbodies attached to objects where the Animator has *Animate Physics* selected now have velocities applied to them when animated. This will give correct physical interactions with other physics objects, and bring the Animator in line with the behaviour of objects animated by the Animation Component.

This will affect how your animated Rigidbodies interact with other Rigidbodies (the Rigidbodies are moved instead of teleported every frame), so make sure to verify that your Animators with **Animate Physics** are behaving as you expect.

Dynamic batching 2D sprites

You should use dynamic batching in upgraded projects that contain 2D sprites. This avoids significant sprite rendering performance issues on devices with Adreno and Mali chipsets.

To use Dynamic Batching, open the PlayerSettings (menu: **Edit** > **Project Settings** > **Player**). Open the **Other Settings** section and, under **Rendering**, tick the **Dynamic Batching** checkbox and untick the **Graphics Jobs (Experimental)** checkbox. Note that these are the default settings for projects created in 5.6.

Graphics Jobs should not affect dynamic batching, but can sometimes cause unexpected behaviors on platforms that use Vulkan and DirectX12.

NUnit library updated

The Unity Test Runner uses a Unity integration of the NUnit library, which is an open-source unit testing library for .NET languages. In Unity 5.6, this integrated library is updated from version 2.6 to version 3.5. This update has introduced some breaking changes that might affect your existing tests. See the NUnit update guide for more information.

---

* 2017–05–19 Page amended with editorial review

Upgrading to Unity 5.5

Leave feedback

Shaders: Physically Based Shading code changes

Physically based shading related code has been refactored in Unity 5.5 (files `UnityStandardBRDF.cginc` and so on). In most cases this does not affect your **shader** code directly, unless you are manually calling some functions directly. Notable changes are listed below.

There are now functions to convert between smoothness, roughness and perceptual roughness: `PerceptualRoughnessToRoughness`, `RoughnessToPerceptualRoughness`, `SmoothnessToRoughness`, `RoughnessToSmoothness`.

The visibility term in `UnityStandardBRDF.cginc` takes `roughness` and not `perceptualRoughness`.

In older versions of Unity, it was possible to do a remapping with Marmoset roughness. With the move to GGX it no longer matches, and `UNITY_GLOSS_MATCHES_MARMOSET_TOOLBAG2` definition and associated code has been removed.

All reads and writes into the Gbuffer should go through new functions `UnityStandardDataToGbuffer` / `UnityStandardDataFromGbuffer`.

Your shader code should call `UnityGlossyEnvironmentSetup()` to initialize a `Unity_GlossyEnvironmentData` struct instead of doing it manually.

The `roughness` variable of `Unity_GlossyEnvironmentData` is actually "perceptual roughness" but it hasn't been renamed to avoid errors with existing shader code. Note: `UnityGlossyEnvironmentSetup` takes `smoothness` as a parameter and calculates perceptual roughness.

The `ndotl` variable value in `UnityLight` is now calculated on the fly and any value written into the variable is ignored.

The `UNITY_GI` macro is deprecated and should not be used anymore.

Shaders: DirectX 9 half-pixel offset issue

Unity 5.5 now handles DX9 half-pixel offset rasterization in the background, which means you no longer need to fix DX9 half-pixel issues either in shaders or in code. See more details in this blog post. If you use any of these checks in your code, they can now be removed:

Checks in shaders for UNITY_HALF_TEXEL_OFFSET and shifting vertices/UVs based on that.
Checks for D3D9 via SystemInfo.graphicsDeviceType or SystemInfo.graphicsDeviceVersion, and shifting vertices/UVs based on that.

The way Unity solves this now is by inserting half-pixel adjustment code into all vertex shaders that are being compiled. As a result, **vertex shader** constant register c255 becomes reserved by Unity, as well as two instructions being added to all shaders, and one more temporary register is used. This should not create problems, unless your vertex shaders use up all the available resources (constant/temporary registers and instruction slots) to the absolute maximum.

Shaders: Z-buffer float inverted

The Z-buffer (depth buffer) direction has been inverted and this means the Z-buffer will now contain 1.0 at the near plane, 0.0 at the far plane. This, combined with the floating point depth buffer significantly increases the depth buffer precision resulting in less Z-fighting and better shadows, especially when using small near planes and large far planes.

Graphics API changes:

Clip space range is [near, 0] instead of [0, far]
_CameraDepthTexture texture range is [1,0] instead of [0,1]
Z-bias is negated before being applied
24 bit depth buffers are switched to 32 bit float format

The following macros/functions will handle reversed Z situations without any other steps. If your shader was already using them, then no changes needed from your side:

Linear01Depth(float z)
LinearEyeDepth(float z)
UNITY_CALC_FOG_FACTOR(coord)
However if you are fetching the Z buffer value manually you will need to do write code similar to:

```
float z = tex2D(_CameraDepthTexture, uv);
#if defined(UNITY_REVERSED_Z)
    z = 1.0f - z;
#endif
```

For clip space depth you can use the following macro. Please **note** that this macro will not alter clip space on OpenGL/ES plaforms but will remain `[-near, far]`:

```
float clipSpaceRange01 = UNITY_Z_0_FAR_FROM_CLIPSPACE(rawClipSpace);
```

_ZBufferParams now contains these values on platforms with a reversed depth buffer. See documentation on platform-specific rendering differences for more information.

```
x = -1+far/near
y = 1
z = x/far
w = 1/far
```

Z-bias is handled automatically by Unity but if you are using a native code **rendering plugin** you will need to negate it in your C/C++ code on matching platforms.

Special Folder: Unity Editor subfolder named "Resources"

All subfolders of the folder named **"Editor"** will be excluded from the build and will not load in Play mode in the Unity Editor. Previously a subfolder named **"Resources"** would have its **assets** included in the build. These **assets** are still loadable by calling Resources.Load() in your Editor **scripts**.

For example, these files are excluded from the build and will **not** load when in Play mode in the Editor, but **will** load if called from scripts:

Editor/Foo/Resources/Bar.png *(this loads from Editor code as "Bar.png")*
Editor/Resources/Foo.png
Editor/Resources/Editor/Resources/Foo.png *(this loads from Editor code as "Foo.png" but not as "Editor/Resources/Foo.png")*
These assets will load in all situations:

Resources/Editor/Foo.png

Resources/Foo/Editor/Bar.png *(this loads as "Foo/Editor/Bar.png")*
Resources/Editor/Resources/Foo.png *(this loads as "Foo.png" and not as "Editor/Resources/Foo.png")*

Backface Tolerance and Final Gather

Previously the 'Backface Tolerance' parameter in Lightmap Parameters was not applied when using final gather for baked GI. It is now applied correctly. The parameter now affects both the realtime GI and baked GI stages (including the final gather).

Affected **scenes** are mainly ones with single sided geometry (like billboards) where it is important to be able to adjust the 'Backface Tolerance' in order to avoid invalidating texels that are seeing the backface of single sided geometry. In scenes that use **billboards** and final gather the **lightmaps** can now be improved by adjusting 'Backface Tolerance', however other scenes might also be affected, if a non-default 'Backface Tolerance' is applied, since it is now correctly accounted for in the final gather stage.

Standard shader BRDF2 now uses GGX approximation

BRDF2, the **standard shader** type set on mobile platforms by default, now uses GGX approximation instead of Blinn-Phong. This makes it look closer to BRDF1 (used on desktops by default) and improves on visual quality.

Should you need to preserve legacy approximation you should modify the BRDF2 code in UnityStandardBRDF.cginc which has the new implementation inside the #if UNITY_BRDF_GGX statement (this is also used by BRDF1 to pick GGX). Change the definition in UnityStandardConfig.cginc or change #if UNITY_BRDF_GGX to #if 0 in the BRDF2_Unity_PBS function.

Gradle for Android

You can now use Gradle to build for Android.

Gradle is not as strict about errors compared with the existing Unity Android build system, meaning that some existing projects may be hard to convert to Gradle. See documentation on Gradle troubleshooting to identify and solve these build failures.

Instantiate Object overload has changed

The specific overload of the Instantiate function that by default, takes a parameter for the original **GameObject** and one for a parent Transform has been changed to work differently. It no longer interprets the original **GameObject**'s position and rotation as a world space position and rotation, thus ignoring the position and rotation of the specified parent Transform.

It now interprets the position and rotation as a local position and rotation within the space of the specified parent Transform, by default. This is consistent with behavior in the Editor. Your scripts will not be automatically updated. This means when you run scripts containing calls to this overload of Instantiate that have not been updated to account for this change, you may experience unexpected behavior.

Renderers and LOD Group components behavior

Disabling a **LOD Group component** no longer disables the Renderers attached to it. The **LOD Group** settings only apply to the Renderers when the **LOD Group component** is enabled. Unity automatically applies this change when you upgrade your project, and the change cannot be reverted.

Upgrading to Unity 5.4

Leave feedback

When upgrading projects from Unity 5.3 to Unity 5.4, there are some changes you should be aware of which may affect your existing project.

Networking: Multiplayer Service API changes

Numerous changes to the Networking API.

Networking: WebRequest no longer experimental

The `WebRequest` interface has been promoted from `UnityEngine.Experimental.Networking` to `UnityEngine.Networking`. Unity 5.2 and 5.3 projects that use `UnityWebRequest` will have to be updated.

Scene View: Tone mapping not automatically applied

An image effect with the `ImageEffectTransformsToLDR` attribute will no longer be applied directly to the **Scene view** if found. A new attribute exists for applying effects to the **Scene view**: `ImageEffectAllowedInSceneView`. The 5.4 **Standard Assets** have been upgraded to reflect this change.

Shaders: Renamed variables

A number of built-in **shader** variables were renamed for consistency:

_Object2World and **_World2Object** are now **unity_ObjectToWorld** and **unity_WorldToObject**
_World2Shadow is now **unity_WorldToShadow[0]**, **_World2Shadow1** is **unity_WorldToShadow[1]** **etc**.
_LightMatrix0 is now **unity_WorldToLight**
_WorldToCamera, **_CameraToWorld**, **_Projector**, **_ProjectorDistance**, **_ProjectorClip** and **_GUIClipTextureMatrix** are now all prefixed with **unity**
The variable references will be automatically renamed in .shader and .cginc files when importing them. However, after the import the shaders will not be usable in Unity 5.3 or earlier, without manually renaming the variables.

Shaders: Uniform arrays

In Unity 5.4, the way arrays of shader properties are handled has changed. Now there is “native” support for float/vector/matrix arrays in shaders (via `MaterialPropertyBlock.SetFloatArray`, `Shader.SetGlobalFloatArray` etc.). These new APIs allow arrays up to 1,023 elements.

The old way of using number-suffixed names for referring to individual array elements is deprecated (e.g. `_Colors0`, `_Colors1`) in both `Material` and `MaterialPropertyBlock`. Properties of this kind, serialized with the `Material` are no longer able to set array elements (but if any uniform array’s name is suffixed by a number it still works).

Shaders: Miscellaneous changes in 5.4

The default shader compilation target has changed to “#pragma target 2.5” (SM3.0 on DX9, DX11 9.3 feature level on WinPhone). You can still target DX9 SM2.0 and DX11 9.1 feature level with “#pragma target 2.0”.

The majority of built-in shaders target 2.5 now: Notable exceptions are Unlit, VertexLit and fixed function shaders. In practice, this means that most of built-in shaders and newly-created shaders, by default, will not work on PC GPUs that are made before 2004. See this blog post for details.

The `Material` class constructor `Material(string)`, which was already deprecated, stops working in 5.4. Using it will print an error and result in the magenta error shader.

The internal shader used to compute screen-space directional light shadows has moved to Graphics Settings. If you were using a customized version of directional light shadows by having a copy of that shader in your project, it will no longer be used. Instead, select your custom shader under **Edit > Project Settings > Graphics**.

Reflection probes share a sampler between the two textures. If you are sampling them manually in your shader, and get an "undeclared identifier samplerunity_SpecCube1" error, you'll need to change code from `UNITY_PASS_TEXCUBE(unity_SpecCube1)` to `UNITY_PASS_TEXCUBE_SAMPLER(unity_SpecCube1,unity_SpecCube0)`.

`UnityEditor.ShaderUtil.ShaderPropertyTexDim` is deprecated; use `Texture.dimension`.

ComputeBuffers

The data layout of ComputeBuffers in automatically-converted OpenGL shaders has changed to match the layout of DirectX ComputeBuffers. If you use ComputeBuffers in OpenGL, remove any code that tweaks the data to match the previous OpenGL-specific layout rules. Please see Compute Shaders for more information.

Playables: Migrating to 5.4

Playables are now structs instead of classes.

The `Playable` structs are handles to native `Playable` classes, instead of pointers to native `Playable` classes.

A non-null `Playable` struct doesn't guarantee that the Playable is usable. Use the `.IsValid` method to verify that your Playable can be used.

Any method that used to return null for empty inputs/outputs will now return `Playable.Null`.

`Playable.Null` is an invalid Playable.

`Playable.Null` can be passed to `AddInput`, `SetInput` and `SetInputs` to reserve empty inputs, or to implicitly disconnect connected inputs.

Using `Playable.Null` or any invalid Playable as an input to any method, or calling a method on an invalid Playable will throw appropriate exceptions for the operation.

Comparing Playables with `null` is now meaningless. Compare with `Playable.Null`.

Playables must be allocated using the `Create` static method of the class you wish to use.

Playables must be de-allocated using the `.Destroy` method on the Playable handle.

Playables that are not de-allocated will leak.

Playables were converted to structs to improve performance by avoiding boxing/unboxing (more info on boxing).

Casting a `Playable` to an object, implicitly or explicitly, will cause boxing/unboxing, thus reducing performance.

Inheriting from a Playable class will cause boxing/unboxing of instances of the child class.

Since only animation is available for now, `ScriptPlayable`s have been replaced by `CustomAnimationPlayables`.

It is no longer possible to derive from base Playables. Simply aggregate Playables in your custom Playables.

Oculus Rift: Upgrading your project from Unity 5.3

Follow these instructions to upgrade your Oculus VR project from Unity 5.3:

Re-enable virtual reality support.

Open the Player Settings. (Menu: **Edit > Project Settings > Player**)
Select **Other Settings** and check the **Virtual Reality Supported** checkbox. Use the Virtual Reality SDK list displayed below the checkbox to add and remove Virtual Reality Devices for each build target.
Remove **Oculus Spatializer**.

Remove the **Oculus Spatializer** Audio Plugin from your project through the Audio Settings Window (Menu: **Edit > Project Settings > Audio**), using the Spatializer Plugin dropdown. It may conflict with the native spatializer and prevent building.

Reordering siblings

There has been a change to the events that are triggered when sibling GameObjects are re-ordered in Unity 5.4. Sibling GameObjects are GameObjects which all share the same parent in the Hierarchy window. In prior versions of Unity, changing the order of sibling GameObjects would cause every sibling to receive an `OnTransformParentChanged` call. In 5.4, the sibling GameObjects no longer get this call. Instead, the parent GameObject receives a single call to `OnTransformChildrenChanged`.

This means that if you have code in your project that relies on **OnTransformParentChanged** being called when siblings are re-ordered, these calls will no longer occur, and you need to update your code to take action when the parent object receives the **OnTransformChildrenChanged** call instead.

This changed because the new behavior gives improved performance.

Rearranging large GameObject hierarchies at runtime

Due to optimisations in the Transform component, using `Transform.SetParent` or `Destroy`ing parts of hierarchies involving 1000+ GameObjects may now take a long time. Calling `SetParent` on or otherwise rearranging such large hierarchies at runtime is not recommended.

Windows Store

The generated Visual Studio project format was updated for all .NET scripting backend SDKs. This fixes excessive rebuilding when nothing has changed in the generated project. You might need to delete your existing generated

*.csproj, especially if it was built with the “Generate C# projects” option checked, so Unity can regenerate it.

Script serialization errors

There are two new script serialization errors to catch when the Unity API is being called from constructors and field initializers during deserialization (loading). Deserialization might happen on a different thread to the main thread and it is therefore not safe to call the Unity API during deserialization. There are more details at the bottom of the (Script Serialization)[script-Serialization] page in the Unity Manual.

Supporting Retina screens

The editor now supports Retina resolutions on Mac OS X with high-resolution text, UI and 3D views.

The Editor GUI is now defined in point space rather than pixel space. On standard resolution displays, there is no change because each point is one pixel. On Retina displays, however, every point is two pixels. The current screen to UI scale is available as `EditorGUIUtility.pixelsPerPoint`. Since Unity can have multiple windows, each on monitors with different pixel densities, this value might change between views.

If your editor code uses regular Editor/GUI/Layout methods, it is quite likely that you will not need to change anything.

If you are using `Screen.width/height`, switch to `EditorWindow.position.width/height` instead. This is because screen size is measured in pixels, but UI is defined in points. For custom Editors displaying in the Inspector, use `EditorGUIUtility.currentViewWidth`, which properly accounts for the presence of a scroll bar.

If you are displaying other content in your UI, such as a RenderTexture, you are probably calculating its size in points. To support Retina resolutions, you will need to convert your point sizes to pixel sizes. There are new methods in EditorGUIUtility for this.

If you are using GUIStyles with custom backgrounds, you can add Retina versions of background textures by putting one texture with exactly doubled dimensions into a ‘GUIStyleState.scaledBackgrounds’ array.

Macs with underpowered graphics hardware may experience unacceptably slow editor frame rates in 3D views due to increased resolution. Retina support can be disabled by choosing “Get Info” on Unity.app in the Finder, and checking “Open in Low Resolution”.

Physics: Meshes and physics transform drift

There are:

Changes to reject physics meshes if they contain invalid (non-finite) vertices.
Changes to avoid physics transform drift by not sending redundant Transform updates.

The 5.3 behavior is that the animation system always sends Transform update messages for animation curves which are constant. These messages wake up the Rigidbodies and fixing this proved very risky in 5.3.

The 5.4 behavior is that if there are no position changes, the Rigidbody does not wake up (as most people would expect).

If your project relies on the 5.3 behavior of waking Rigidbody all the time, it might not work as expected in 5.4.

Web Player

The Unity Web Player target has been dropped from Unity 5.4. If you upgrade your project to 5.4, you will not be able to deploy it to the Web Player platform.

If you have legacy Web Player projects that you need to maintain, do not upgrade them to 5.4 or later.

5.4 Networking API Changes

Leave feedback

In Unity 5.4 we made a number of changes to the matchmaking API. Our intent was to simplify and clean up the API.

If you used the matchmaking API in an earlier version of Unity, you will need to check and update the classes and functions listed below.

MatchDesc has been renamed to **MatchInfoSnapshot**.

All request and response classes are removed, so there are no longer any overloaded functions in NetworkMatch. Instead we updated the parameter list of all functions to accommodate the loss of the missing classes and we updated the 2 delegates.

Setting up

```
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.Networking.Match;
NetworkMatch matchMaker;
void Awake()
{
    matchMaker = gameObject.AddComponent<NetworkMatch>();
}
```

CreateMatch (Before 5.4)

```
CreateMatchRequest create = new CreateMatchRequest();
...
matchMaker.CreateMatch(create, OnMatchCreate);
```

Or

```
matchMaker.CreateMatch("roomName", 4, true, "", OnMatchCreate);
```

Is now:

```
matchMaker.CreateMatch("roomName", 4, true, "", "", "", 0, 0, OnMatchCreate);
```

CreateMatch Callback (Before 5.4)

```
public void OnMatchCreate(CreateMatchResponse matchResponse)
{
    ...
}
```

Is now:

```
public void OnMatchCreate(bool success, string extendedInfo, MatchInfo matchInfo
{
    ...
}
```

ListMatches (Before 5.4)

```
ListMatchRequest list = new ListMatchRequest();

matchMaker.ListMatches(list, OnMatchList);
```

Or

```
matchMaker.ListMatches(0, 10, "", OnMatchList);
```

Is now:

```
matchMaker.ListMatches(0, 10, "", true, 0, 0, OnMatchList);
```

ListMatches Callback (Before 5.4)

```
public void OnMatchList(ListMatchResponse matchListResponse)
{
    ...
}
```

Is now:

```
public void OnMatchList(bool success, string extendedInfo, List<MatchInfoSnapsho
{
    ...
}
```

JoinMatch (Before 5.4)

```
JoinMatchRequest join = new JoinMatchRequest();

matchMaker.JoinMatch(join, OnMatchJoined);
```

Or

```
matchMaker.JoinMatch(match.networkId, "", OnMatchJoined);
```

Is now:

```
matchMaker.JoinMatch(networkId, "" , "", "", 0, 0, OnMatchJoined);
```

JoinMatch Callback (Before 5.4)

```
public void OnMatchJoined(JoinMatchResponse matchJoin)
{
    ...
}
```

Is now:

```
public void OnMatchJoined(bool success, string extendedInfo, MatchInfo matchInfo
{
    ...
}
```

DestroyMatch (Before 5.4)

```
DestroyMatchRequest destroy = DestroyMatchRequest();

matchMaker.DestroyMatch(destroy, OnMatchDestroy);
```

Or

```
matchMaker.DestroyMatch(netId, OnDestroy);
```

Is now:

```
matchMaker.DestroyMatch(netId, 0, OnMatchDestroy);
```

DestroyMatch Callback (Before 5.4)

```
public void OnMatchDestroy(BasicResponse response)
{
    ...
}
```

Is now:

```
public void OnMatchDestroy(bool success, string extendedInfo)
{
    ...
}
```

DropConnection (Before 5.4)

```
DropConnectionRequest drop = DropConnectionRequest();

matchMaker.DropConnection(drop, OnMatchDropConnection);
```

Or

```
matchMaker.DropConnection(netId, nodeId, OnMatchDropConnection);
```

Is now:

```
matchMaker.DropConnection(netId, nodeId, 0, OnMatchDropConnection);
```

DropConnection Callback (Before 5.4)

```
public void OnMatchDropConnection(BasicResponse response)
{
    ...
}
```

Is now:

```
public void OnMatchDropConnection(bool success, string extendedInfo)
{
    ...
}
```

Upgrading to Unity 5.3

Leave feedback

Global Illumination

Lightmap Snapshot was renamed to Lighting Data **asset**. The internal format of the lighting data was changed after the upgrade to **Enlighten** 3. Snapshots from a previous versions of Unity are no longer supported and should be rebaked.

This also affects streamed **scene** AssetBundles with realtime GI. **Lightmap** data won't be loaded and thus such bundles should be also rebuilt.

Light probes and environment lighting is now consistent in gamma and linear color space. Some differences in environment lighting compared to Unity 5.2 is to be expected. Output is matching Unity 4.x intensity wise now but since 4.x and our light projection code generates L2 coefficients and Enlighten only outputs L1, the falloff for the final **light probes** can appear different. L2 support for **light probes** will appear in a future release. Directional non-important lights should now match 4.x. Light probes are always passed to the shaders in linear color space and final gamma conversion happens on the GPU. If you are using Unity's ShadeSHxxx functions for evaluating the spherical harmonics in the **shader**, you should not have to change your shaders. In UNITY_STANDARD_SIMPLE shaders the SH evaluation is not split between the **pixel** and **vertex shader**, so we limit the linear to gamma conversions to only happen once and only in the **vertex shader**. On more advanced GPUs the calculation is split between vertex and **fragment shader**.

Shuriken

The particle size in the **Collision** Module has been replaced by a new parameter: Radius Scale. This new parameter acts as a multiplier on the actual particle size. If you were using the old value to do anything other than approximate the particle sizes, then you will need to reconfigure your **collision** bounds using the new parameter.

Multi Scene Editing

The multi scene editing feature introduces new API through EditorSceneManager and SceneManager. Which means that many of the API's on EditorApplication and Application has been deprecated.

EditorApplication.NewScene
EditorApplication.NewEmptyScene
EditorApplication.OpenScene
EditorApplication.OpenSceneAdditive
EditorApplication.SaveScene
EditorApplication.SaveCurrentSceneIfUserWantsTo
EditorApplication.SaveCurrentSceneIfUserWantsToForce
The above all have equivalent APIs on EditorSceneManager

EditorApplication.currentScene
Internally this will return the name of the active scene in the Scene Manager, but to get all currently open scenes use the EditorSceneManager APIs

EditorApplication.MarkSceneDirty
EditorApplication.isSceneDirty
Each scene now has its own dirty flag. Get the scenes through the EditorSceneManager and check their state. Setting scenes dirty is also done through the EditorSceneManager. The deprecated APIs all operate on the active scene only.

Application.LoadLevel
Application.LoadLevelAsync
Application.LoadLevelAdditive
Application.LoadLevelAdditiveAsync
Application.LoadLevel[Async](path) redirects to SceneManager.LoadScene[Async](path, false) and Application.LoadLevelAdditive[Async](path) redirects to SceneManager.LoadScene[Async](path, true)

Application.loadedLevel
Application.loadedLevelName

These respectively gets the Build Setting Index of the active scene and the name of the active scene. You should use the SceneManager to get the indices and names of all loaded scene.

Note also that EditorApplication.OpenSceneAdditive can nolonger be called during play in the Editor. That also means it cannot be called from a [PostprocessScene] callback. If EditorApplication.OpenSceneAdditive is called during play anyway, then play mode will be stopped.

Precompiled Shader Assets

Precompiled shader assets are no longer supported - this means you can no longer click “show compiled code” and copy the resulting disassembly into a new shader asset. Old shader assets that are precompiled will be marked as unsupported.

The “show compiled code” in the shader **inspector** will still work and will display the disassembly of the shader on each platform.

Likewise you can still view the generated code for a **surface shader**, modify it, and copy that into a new shader asset - since it is only the HLSL source you are modifying.

This will affect AssetBundles built in previous versions of Unity - they have compiled shader assets inside them by definition. Any shaders in such bundles will need to be rebuilt.

For more detailed information you can see this unity blog post about upcoming feature deprecation.

OpenGL 4.x support on desktop

As a new feature, the OS X Editor and Standalone now support the new GL backend, which enables the use of OpenGL 3.x and 4.x features such as tessellation and geometry shaders. However, as Apple restricts the OpenGL version on OS X desktop to 4.1 at most, it does not support all DirectX 11 features (such as Unordered Access Views or Compute Shaders). This means that all shaders that are configured to target Shader Level 5.0 (with #pragma target 50) will fail to load on OS X.

Therefore a new shader target level is introduced: #pragma target gl4.1. This target level requires at least OpenGL 4.1 or DirectX 11.0 Shader Level 5 on desktop, or OpenGL ES 3.1 + Android Extension Pack on mobiles.

AssetBundles

AssetBundle’s container format was changed in order to support new LZ4 **compression** and have a basis for further improvements. Bundles created in earlier version (2.x, 3.x) are deprecated and not supported. Bundles created in Unity 4.x, 5.0–5.2 are supported and could be loaded. But, if they were already cached on a user device using WWW.LoadFromCacheOrDownload method, they will be redownloaded. Also take in mind that data in such bundles might be a subject to change (see e.g. Global Illumination section).

GetComponent(s)InChildren

GetComponentsInChildren has a slightly changed behaviour in the case where you invoke it on a **gameObject** who has a parent that is inactive. Previously you would always get an empty array as a result. Because that is never what you want, and because that meant GetComponentsOnChildren didn’t work on **prefabs**, this has been changed to ignore any active state from the target game object’s parents. Also, the singular version GetComponentInChildren() now has an optional includeInactive argument.

UI/default shaders

Using a default new **UI** shader on non new **UI** objects is not longer supported by default. Previously there was an ‘if’ check to determine whether or not the _clipRect should be used but for performance reasons this check was removed. To continue using a new **UI** shader on a non new **UI** object you will need to specify a valid clipRect yourself.

Point and spot shadow casting lights

Point lights that are selected to cast shadows now have a working Bias slider, to allow adjustment and balancing of shadowing artifacts (under shadowing vs. shadow acne). This means any existing point lights which may have had a Bias set before that wasn’t doing anything will now start having an effect, and this will change the shadow casting behaviour.

Spot lights that cast shadows now have a new slider which allows you to select the near clip distance. This is the distance to the light below which any objects won't cast shadows. Low values include close-by objects, at the cost of greatly reduced precision for the shadows. In previous Unity versions this was calculated at 4% of the total range of the light, which could be much too high for large lights. Now it defaults to 0.2, which should work for most cases.

Quaternion Mathematics

The new support of importer Euler rotation curves, and the support of all the different Euler rotation orders necessitated a rewrite of the QuaternionToEuler and EulerToQuaternion mathematics functions, both in traditional and SIMD versions. Those new variations have not been made available in the API yet, and are only used internally for now.

This should have very little impact, but there are minute differences (<0.01 degrees) in the results between the previous version and the new one, and only when very close to gimbal lock conditions. Tests run have shown the new version to be more accurate most of the time, and the average error to be smaller by at least a factor of 10.

JointDriveMode flags

JointDriveMode flags are now obsolete, and thus have been removed. However, in earlier versions of Unity they were incorrectly being used to ignore the Configurable **Joint**'s **Joint** Drive stiffness and damping settings. When upgrading a project to Unity 5.3 which uses **Configurable Joints**, users should be aware that these settings may now be having an effect when previously they did not - because they were wrongly being ignored based on the old JointDriveMode flags.

Legacy Light Animation

As of 5.3, Legacy Animations, both existing and new, will not animate Light properties. Changes to the underlying data structure of the Lights have made them incompatible with Legacy. To properly animate Lights, please use the Animator **Component**.=======

Editor Extensions

The scene's dirty flag is now respected when saving scenes. Editor extensions that do not correctly set the dirty flag may fail to save data correctly. Use Undo.RecordObject to record that an object is about to change and to update the scene's dirty flag accordingly, or EditorSceneManager.MarkSceneDirty to forcibly mark the entire scene as dirty.

Camera Depth Texture shader variable

The `_CameraDepthTexture` shader variable has been fixed to consistently refer to the primary depth texture on the **camera**, and not as it did previously to the last depth texture rendered by any **camera**. If you are **rendering** a secondary **camera** in a script to e.g. obtain a half-resolution depth buffer and you need to bind its depth texture, you should now use the `_LastCameraDepthTexture` variable which has the semantics of referring to the depth texture of whichever camera rendered last.

ComputeBuffers

The data layout of ComputeBuffers in automatically-converted OpenGL shaders has changed to match the layout of DirectX ComputeBuffers. If you use ComputeBuffers in OpenGL, remove any code that tweaks the data to match the previous OpenGL-specific layout rules. Please see Compute Shaders for more information.

Upgrading to Unity 5.2

Leave feedback

Global Illumination

UV packing for baked UVs not filling the 0–1 space (smaller or bigger) has been fixed. It makes the resolution assigned to each object work much more reliably if that object's unwrap is not filling the 0–1 space and also when its bounds are non-square. Please review the resolution on your instances for baked **lightmaps**.

Shader variant stripping was fixed for realtime lightmaps. Now each lightmaps mode (non-directional, directional and directional specular) variant can be picked for baked and realtime GI separately. Please review your settings if you previously selected a specific lightmaps mode variant in the Graphics Settings to make that mode work for realtime lightmaps.

Bounce scale has been changed from the arbitrary value of 0.7 to 1.0. The bounce is the product of the albedo and bounce scale. Artists should set real-life albedo values (the brightest non-metallic is snow with 0.9). This is our PBS reference http://forum.unity3d.com/threads/official–5–0-pbr-calibration-charts.289416/

Since you should author physically correct albedo, it makes sense for us to set the scale close to 1. We already clamp albedo values in the meta pass, so the bounce scale should just be 1.0f.

Please note that if you choose to set albedo to 1.0 in a custom meta pass without clamping, then the **scene** can look like it's exploding with light.

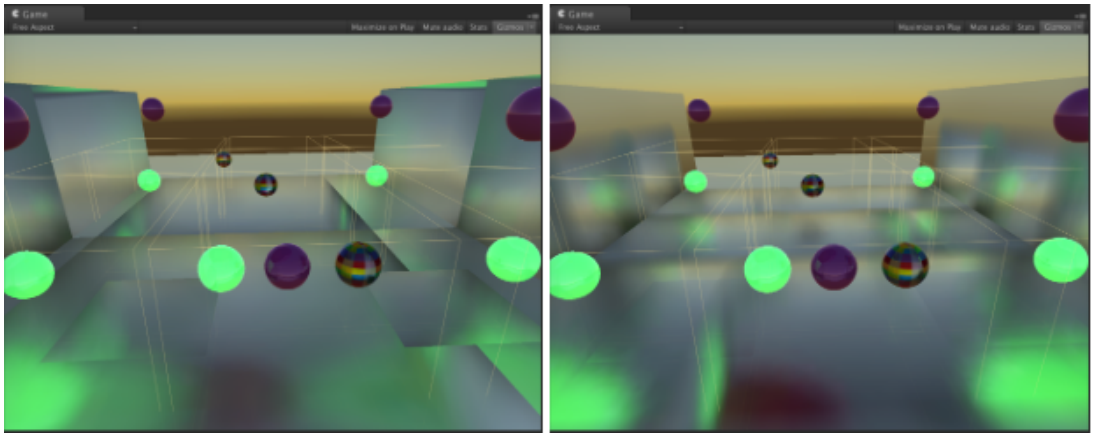
Shaders

"Fixed Function" style **shaders** (the ones that use SetTexture, Lighting On etc.) internally get turned into actual shaders at shader import time now. Upside is that they now work on all platforms (previously did not work on consoles), and with more consistency. Also a lot of code and fixed function related inefficiencies got removed from runtime, making **rendering** a bit faster. Downside is, creating fixed function shaders at runtime - using `new Material(fixedFunctionShaderString)` - does not work anymore. That constructor was deprecated in Unity 5.1, and now in 5.2 it actually stopped working for fixed function shaders.

Reflection Probes

We've changed how **Reflection Probes** are rendered when using **Deferred Shading**, in order to allow "screen space reflections" effects in the future. Short version is: in **deferred shading**, **reflection probes** are per-pixel instead of per-object now.

*Comparison of current behavior (reflection probes per object; in some cases hard to avoid harsh reflection transitions between large objects) and reflection probes per **pixel** (transitions much less visible; and they happen at probe boundaries not at object boundaries):*



Before (5.0 and 5.1)

Reflection probes are sampled during the G-buffer pass, in exactly the same way as in **forward rendering**. They are written into "emission" buffer together with **light probes**, lightmaps and emissive material parts.

This meant you get one (or two, when probe blending is on) reflection probes *per object*.

Reflections being together with emission/lightmaps in the same buffer means that doing SSRR "properly" is hard. SSRR provides reflections by itself (and falls back to reflection probes where it can't), but it does not know which part of "emission buffer" color is coming from reflection probes.

Now (5.2)

When using deferred shading, do not sample reflection probes during G-buffer pass.

Instead, after the G-buffer is done, make a separate "deferred reflections" pass that draws reflection probes as boxes in screenspace; that output reflection information into a separate render target.

[Future: SSRR effect will use this separate reflections buffer]

Combine reflections buffer & emission buffer at the end.

What does this mean? *(everything below only affects deferred shading)*

Reflection probes are no longer per-object; they are effectively per-pixel. It is easier to have large objects affected by many reflection probes.
Also probes got a "blend distance" which defines how much space around the probe is used for blending into other probes.
Smaller probes "override" larger ones.
Reflection probe Renderer flags (probe blending, etc.) are ignored; everything is affected by reflection probes in the same way (since it happens in screenspace now). This is very similar to how "receive shadows" flag is ignored in deferred shading.

Your custom-written shaders that do deferred shading should mostly continue to work (in the worst case, they will be sampling a black reflection cubemap). If you do some totally crazy stuff in shaders, they might need to be updated to work with deferred shading in 5.2.
If you are using custom deferred shading light pass shader (with custom BRDF etc.), you'll probably want to use custom deferred reflections shader too, with the same BRDF applied to reflection probes.

Shuriken

Particles are now generated in world space, which may require an update to any custom **vertex shaders**. This change was made in order to allow re-use of the particle buffers between each eye for VR.

Mesh particles now support the Texture Sheet Animation module. It's worth checking that your existing effects do not have this enabled by accident, otherwise you may see a change in behaviour.

The Dampen parameter in the Limit Velocity over Lifetime module used to have a stronger effect at higher framerates. This has been fixed, and if your game is targetting 30fps, your old effects will be unaffected by this change. However, if your game targets a differnet **FPS**, you can update the Dampen value using this formula, to ensure your effect is unchaged in 5.2:

```
newDampen = 1.0f - pow(1.0f - oldDampen, targetFPS / 30.0f);
```

Graphics (Other items)

`Material.CopyPropertiesFromMaterial` now also copies shader keywords and render queue. If you were relying on them being not copied, you'll have to change your code.

SpeedTree materials now need to be re-generated as there are changes to SpeedTree built-in shaders. You could do so by selecting SpeedTree prefabs in your project and hit "Apply & Generate Materials" button. *Please be noted that by doing so your customisations to the generated material assets (if any) will be overwritten.*

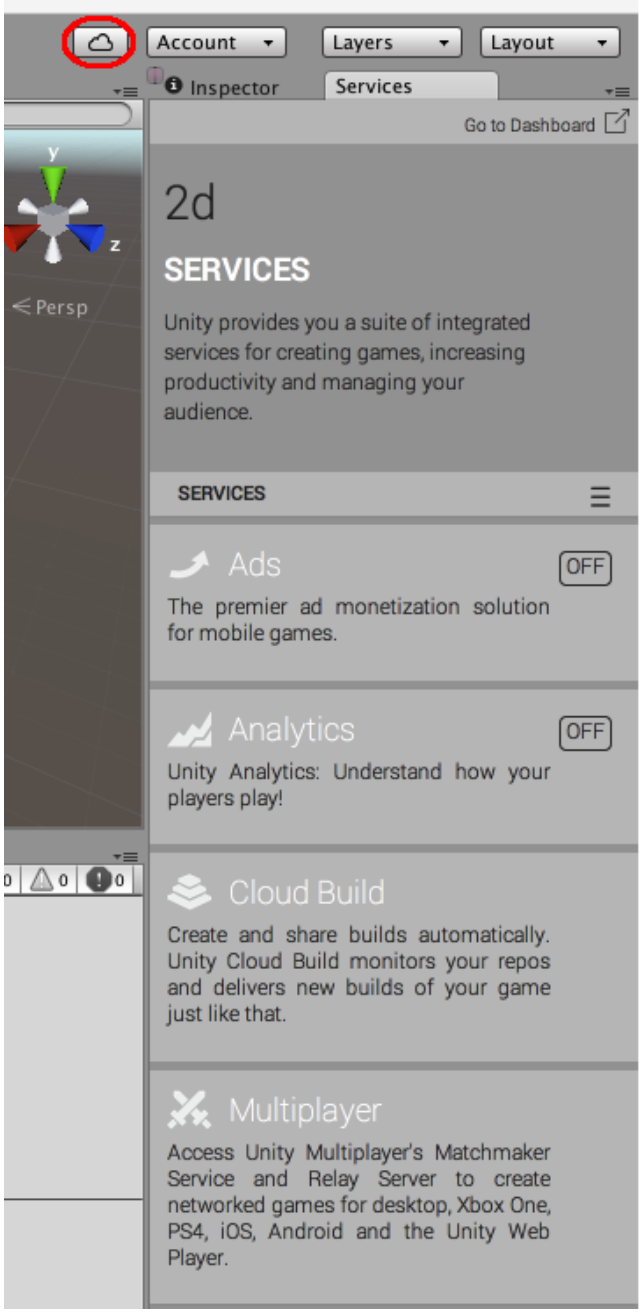
UI

In 5.2 we have combined the shaders that text and normal UI element rendering users. A side effect of this is that if you specify a manual font texture in a 32bit format then the color channels will be honored. This means that black texture channels will result in black text where previously the text would be white (we only looked at the alpha). If you wish to use custom textures for your fonts do one of the following:

Change the import format of the texture to A8. This will only keep the alpha component and Unity will generate the text as white by default.
Specify a color / colors in the texture for Unity to use when rendering the text

Multiplayer

The way the project identification is handled has changed in Unity 5.2, now the project is automatically registered and you do not need to manually enter an ID anywhere. There is a Multiplayer panel in the Services Window (open it with cloud icon in upper right corner) and in there you can find a deep link directly to the project on the website (Go to dashboard). When configured the Multiplayer configuration will appear here.



Upgrading to Unity 5.0

Leave feedback

Lots of Unity's features and systems have been upgraded in Unity 5, including Physics, Audio, Animation, **Shaders** and more. While many areas are upgraded automatically when you open an older project, there are certain areas where you may need to manually adjust or refactor parts of your project.

These pages are a quick reference to most of the changes that you need to be aware of when upgrading a project from Unity 4 to Unity 5:

AI in Unity 5.0
Animation in Unity 5.0
Audio in Unity 5.0
Baked Data in Unity 5.0
Plugins in Unity 5.0
Physics in Unity 5.0
Shaders in Unity 5.0
Other Upgrade Notes for Unity 5.0

AI in Unity 5.0

Leave feedback

These are notes to be aware of when upgrading projects from Unity 4 to Unity 5, if your project uses the AI/ **Navmesh** features.

Navmesh contour may look different due to changed partitioning - in cases with narrow corridor/doorways or similar - this can lead to a difference in connectivity. Solve the issue by tweaking the **voxel** size for navmesh building.

Setting the destination for a NavMeshAgent doesn't resume the agent after calling 'Stop' - call 'Resume' explicitly to resume the agent.

NavMeshAgent.updatePosition: When updatePosition is false and the agent transform is moved the agent position doesn't change. Previously the agent position would be reset to the transform position - constrained to the nearby navmesh.

NavMeshObstacle **component**: The default shape for newly created NavMeshObstacle components is a box. The selected shape (box or capsule) now applies to both carving and avoidance.

Navmesh built with earlier versions of Unity is not supported. You must rebuild with Unity 5. You can use the following script as an example how to rebuild NavMesh data for all of you **scenes**.

Rebake script example

```
#if UNITY_EDITOR
using System.Collections.Generic;
using System.Collections;
using System.IO;
using UnityEditor;
using UnityEngine;
using UnityEngine.AI;
public class RebakeAllScenesEditorScript
{
    [MenuItem ("Upgrade helper/Bake All Scenes")]
    public static void Bake()
    {
        List<string> sceneNames = SearchFiles (Application.dataPath, "*.unity");
        foreach (string f in sceneNames)
        {
            EditorApplication.OpenScene(f);

            // Rebake navmesh data
            NavMeshBuilder.BuildNavMesh ();

            EditorApplication.SaveScene ();
        }
    }
```

```
    static List<string> SearchFiles(string dir, string pattern)
    {
        List <string> sceneNames = new List <string>();
        foreach (string f in Directory.GetFiles(dir, pattern, SearchOption.AllDi
        {
            sceneNames.Add (f);
        }
        return sceneNames;
    }
}
#endif
```

Animation in Unity 5.0

Leave feedback

These are notes to be aware of when upgrading projects from Unity 4 to Unity 5, if your project uses animation features.

Asset Creation API

In 5.0 we introduced an API that allows you to build and edit Mecanim **assets** in the editor. For users that have previously used the unsupported API (in UnityEditorInternal namespace) you will need to manually update your **scripts** to use the new API.

Here is a short list of the most encountered type changes :

Previous:	*New:*
UnityEditorInternal.BlendTree	UnityEditor.Animations.BlendTree
UnityEditorInternal.AnimatorController	UnityEditor.Animations.AnimatorController
UnityEditorInternal.StateMachine	UnityEditor.Animations.AnimatorStateMachine
UnityEditorInternal.State	UnityEditor.Animations.AnimatorState
UnityEditorInternal.AnimatorControllerLayer	UnityEditor.Animations.AnimatorControllerLayer
UnityEditorInternal.AnimatorControllerParameter	UnityEditor.Animations.AnimatorControllerParameter

Also note that most accessor functions have been changed to arrays:

```
UnityEditorInternal.AnimatorControllerLayer layer = animatorController.GetLayer(inde
```

becomes:

```
UnityEditor.Animations.AnimatorControllerLayer layer = animatorController.layers[inc
```

A basic example of API usage is given at the end of this blog post: http://blogs.unity3d.com/2014/06/26/shiny-new-animation-features-in-unity–5–0/

For more details refer the the Scripting API documentation.

Audio in Unity 5.0

Leave feedback

These are notes to be aware of when upgrading projects from Unity 4 to Unity 5, if your project uses audio features.

AudioClips

A number of things has changed in the AudioClip. First, there is no longer a 3D flag on the **asset**. This flag has been moved into the AudioSource in form of the Spatial Blend slider allowing you to, at runtime, morphing sounds from 2D to 3D. Old projects will get imported in such a way that AudioSource's on **GameObjects** in the **scene** that have a clip assigned will get their Spatial Blend parameter set up according to the old 3D flag of the AudioClip. For obvious reasons this is not possible for **scripts** that dynamically assign clips to sources, so this requires manual fixing.

While the default setting for the old 3D property was true, by default in the new system, the default of the Spatial Blend parameter is set to 2D.

Finally, AudioClips can now be multi-edited.

Format

The naming of the Format property has changed so that it reflects the method by which the data is stored rather than particular file format which deviates from platform to platform. So from now on Uncompressed refers to raw sample data, Compressed refers to a lossy **compression** method best suited to the platform and ADPCM refers to a lightweight (in terms of CPU) compression method that is best suited to natural audio signals which contain a fair amount of noise (footsteps, impacts, weapons etc) and are to be played in large amounts.

Preloading and loading audio data in the background

A new feature of AudioClips is that they provide support for an option for determining whether to preload the audio data or not. Any property of the AudioClip is detached from the actual audio data loading state and can be queried at any time, so having the possibility to load on demand now helps keeping the memory usage of AudioClips low. Additionally to this AudioClips can load their audio data in the background without blocking the main game thread and causing frame drops. The load process can of course be controlled via the scripting API.

Multi-editing

All AudioClips now support multi-editing.

Force to Mono

The Force To Mono option now performs peak-normalization on the resulting down mix.

GetData/SetData

These API calls are only supported by clips that are either storing the audio data uncompressed as PCM or perform the decompression at on load. In the past more clips supported this, but the pattern wasn't very clear since it depended both on the target platform and had different behavior in the editor and standalone players. As a new thing, tracker files can be decompressed as PCM data into memory too, so GetData/SetData can also be used on these.

AudioSource pause behaviour

Pausing in Unity5 is now consistent between Play calls and PlayOneShots calls. Pausing an AudioSource pauses voices played both ways, and calling Play or PlayOneShot unpauses the AudioSource for voices played both ways too.

To assist with unpausing an AudioSource without playing the assigned clip, (useful for when there are oneshot voices playing), we have added a new function AudioSource.Unpause ().

Audio mixer

The AudioMixer is a new feature of Unity 5 allowing complex routing of the audio data from AudioSource's to mix groups where effects can be applied. One key difference from the **Audio Filter components** is that audio filters get instantiated per AudioSource and therefore are more costly in terms of CPU if a game has a large number of AudioSources with filters or if a script simply creates many instances of a GameObject containing . With the mixer it is now possible to set up a group with the same effects and simply routing the audio from the AudioSource through the shared effects resulting in lower CPU usage.

The mixer does not currently support script-based effects, but it does have a new native audio **plugin** API allowing developers to write high-performance effects that integrate seamlessly with the other built-in effects.

AudioSettings

The way the audio system is configured has changed. The overall settings for setting speaker mode and DSP buffer size (i.e. latency) should still be configured in the audio **project settings**, and additional to these it is now also possible to configure the real and virtual voices counts. The real voice count specifies how many voices can be heard concurrently and therefore has a strong impact on the overall CPU consumption of the game. Previously this was hardcoded to 32 with some platform-specific exceptions. When the number of playing voices exceeds this number, those that are least audible will be put on hold until these voices become dominant or some of the dominant voices stop playing. These voices are simply bypassed from playing. They are not stopped, simply made inactive until there is bandwidth again. The virtual voice count defines how many voices can be managed in total, so if the number of voices playing exceeds this, the least audible voices will be stopped.

AudioSettings.outputSampleRate and AudioSettings.speakerMode can still be read from, but the setters are now deprecated, as is the SetDSPBufferSize function. As a replacement for these for audio configuration changes that need to happen at runtime there is now a structure called AudioConfiguration. This can be obtained via AudioSettings.GetConfiguration() which will return the active settings on the audio output device. Changes can be made to this structure and applied via AudioSettings.Reset() which will return a boolean result depending on the successful or failed attempt of applying the changes.

Whenever changes to the audio configuration happen, either caused by scripts via AudioSettings.Reset() or because of external events such as plugging in HDMI monitors with audio support, external sound cards or USB headsets, a user-defined callback AudioSettings.OnAudioConfigurationChanged(bool deviceChanged) will be invoked. Its argument deviceChanged will be false if the change was caused by an AudioSettings.Reset() call, and true if it was caused by an external device change (this could also be changing the sample rate of the audio device in use). The callback allows you to recreate any volatile sounds such as generated PCM clips, restore any audio state, or further adapt the audio settings through a call to AudioSettings.Reset().

Baked Data in Unity 5.0

Leave feedback

When upgrading a project from Unity 4 to Unity 5, you may need to re-bake stored data because some of the prebaked data formats have changed.

Baked **Occlusion Culling** data format was changed. Rebaking is required for the **occlusion culling** data.

Baked Lighting data format was changed. Rebaking is required for the lighting data.

Plugins in Unity 5.0

Leave feedback

These are notes to be aware of when upgrading projects from Unity 4 to Unity 5, if your project uses **plugins**, including native audio **plugins**.

Plugins are no longer required to be placed in Assets\Plugins<Platform> folders, they now have settings for checking which platforms are compatible and platform specific settings (like setting compatible CPU, SDK's) etc. By default new plugins are marked as compatible with 'Any Platform'. To make transition easier from older Unity versions, Unity will check in what folder plugin is located, and set initial settings accordingly, for ex., if plugin is located in Assets\Plugins\Editor, it will be marked as compatible with Editor only, if plugin is located in Assets\Plugins\iOS, it will be marked as compatible with iOS only, etc. Also please refer to 'PluginInspector documentation' for more info.

Native Plugins in the Editor

32-bit native plugins will not work in the 64-bit editor. Attempting to load them will result in errors being logged to the console and exceptions being thrown when trying to call a function from the **native plugin**.

To continue to use 32-bit native plugins in the editor, use the 32-bit editor version (provided as a separate installer).

To have both 32-bit and 64-bit versions of the same plugin in your project, simply put them in two different folders and set the supported platforms appropriately in the importer settings.

Restrictions:

We currently do not provide a 32-bit editor for OSX.

Native Audio Plugins

Plugins with the prefix "audioplugin" (case-insensitive) will be automatically loaded upon scanning so that they are usable in the audio mixer. The same applies to standalone builds where these plugins are loaded at startup time before any **assets** have yet been loaded, as certain **assets** such as audio mixers require them to be loaded in order to instantiate effects.

Physics in Unity 5.0

Leave feedback

Unity 5.0 features an upgrade to PhysX3.3 SDK. Please give this blogpost a quick look before taking any action on your 4.x projects. It should give you a taste of what to expect from the new codebase: High Performance Physics in Unity 5. Please be warned that PhysX3 is not 100% compatible with PhysX2 and requires some actions from the user when upgrading.

General overview of updates

Unity 5.0 physics could be expected to work up to 2x faster than in previous versions. Most of the components you were familiar with are still there, and you will find them working as before. Of course, some behaviours were impossible to get the same and some were just weird behaviours caused by limitations of the pre-existed codebase, so we had to take changes. The two areas that got the most significant change are Cloth **component** and WheelCollider **component**. We're including a section about each of them below. Then, there are smaller changes all over the physics code that cause incompatibility.

Changes that are likely to affect projects:

Adaptive force is now switched off by default

Adaptive force was introduced to help with the simulation of large stacks, but it turned out to be great only for demos. In real games it happened to cause wrong behaviour. You can switch it on in the editor physics properties: Edit -> Project settings -> Physics -> Enable adaptive force.

Smooth sphere collisions are removed both from terrain and meshes.

PhysX3 has the feature that addresses the same issue and it's no longer switchable as it's considered to be a solution without major drawbacks.

Springs expose larger amplitudes with PhysX3.

you may want to tune spring parameters after upgrading.

Setting Terrain Physics Material has changed

Use TerrainCollider.sharedMaterial and TerrainCollider.material to specify physics material for **terrain**. The older way of setting physics material through the TerrainData will no longer work. As a bonus, you can now specify **terrain** physics materials on a per **collider** basis.

Shape casting and sweeping has changed:

Shape sweeps report all shapes they hit (i.e CapsuleCast and SphereCast would return all shapes they hit, even the ones that fully contain the primitive)
Raycasts filter out shapes that contain the raycast origin

Compound Collider events:

When using compound colliders, OnCollisionEnter is now called once per each contact pair

Triggers must be convex:

From now on, you can have triggers only on convex shapes (a PhysX restriction):

TerrainCollider no longer supports IsTrigger flag
MeshCollider can have IsTrigger only if it's convex

Dynamic bodies must be convex:

Dynamic bodies (i.e. those having Rigidbody attached with Kinematic = false) can no longer have concave **mesh** colliders (a PhysX limitation).

If you want to have **collisions** with concave meshes, you can only have them on static colliders and kinematic bodies

Ragdoll joints:

Joint setups for ragdolls will likely need tweaking.

These suggestions apply to **joints** in general as well.

See the Joint And Ragdoll Stability page for most recent version of this guide.

Avoid small angles of “Angular Y Limit” and “Angular Z Limit”. Depending on your setup the minimum angles should be around 5 to 15 degrees in order to be stable. Instead of using a small angle try setting the angle to zero. This will lock the axis and provide a stable simulation.

Set “Enable Preprocessing” to false (unchecked). Disabling the preprocessing can help against joints “blowing up”. Joints can “blow up” if they are forced into situations where there is no possible way to satisfy the joint constraints. This can occur if jointed rigid bodies are pulled apart by static collision geometry, like spawning a ragdoll partially inside a wall.

Ragdoll stability or stretching: If ragdolls are given extreme circumstances, such as spawning partially inside a wall or pushed with a large force, the joint solver will be unable to keep the rigid bodies together. This can result in stretching or a “cloud of body parts”. Please enable projection on the joints, using either “ConfigurableJoint.projectionMode” or “CharacterJoint.enableProjection”.

If bodies connected with joints are jittering try increase Edit->Project Settings->Physics->“Solver Iteration Count”. Try between 10 or 20.

Never use direct transform access with kinematic bodies jointed to other bodies. Doing so skips the step where PhysX computes internal velocities of corresponding bodies and thus makes solver provide unpleasant results. We’ve seen some 2D projects using direct transform access to flip characters via altering transform.direction on the root boon of the rig. This behaves much better if you use MovePosition / MoveRotation / Move instead.

Rigidbody’s constraints are applied in local space

The locking mechanism we used in Unity 4 was basically discarding changes to locked rotations & was resetting the angular speeds as a post-solver step. This was working mostly fine except that there were issues with bodies failing to go asleep as the solver wanted to adjust the rotations every frame; and a few related cases were noticed over the years. When working on PhysX3 integration we utilised this new feature of PhysX 3.3 where we can set the infinite inertia tensor components for locked rotational degrees of freedom. This is now supported in the solver so the body would now go to sleep appropriately, but since this is inertia, it is applied in local coordinates.

WheelCollider

The new WheelCollider is powered by the PhysX3 Vehicles SDK that is basically a completely new vehicle simulation library when compared to the code we had with PhysX2.

Read more about the new WheelCollider here.

Cloth

Unity 5 uses the completely rewritten cloth solver provided by the new PhysX SDK. This cloth solver has been designed with character clothing in mind, and is a great improvement compared to the old version in terms of performance and stability. Unity 5 replaces the SkinnedCloth and InteractiveCloth components in Unity 4 with a single Cloth component, which works in conjunction with a SkinnedMeshRenderer. The functionality is similar to the previous SkinnedCloth component, but it is now possible to assign arbitrary, non-skinned meshes to the SkinnedMeshRenderer, so you can still handle cloth simulation on any random mesh.

However, some functionality which was available on the old InteractiveCloth is now no longer supported by the new version of PhysX as it is difficult to implement these with good performance. Specifically:

you can no longer use cloth to collide with arbitrary world geometry
tearing is no longer supported
you can no longer apply pressure on cloth
you can no longer attach cloth to colliders or have cloth apply forces to **rigidbodies** in the **scene**.

Shaders in Unity 5.0

Leave feedback

These are notes to be aware of when upgrading projects from Unity 4 to Unity 5, if your project uses custom **shader** code.

Shaders no longer apply a 2x multiply of light intensity

Shaders no longer apply a 2x multiply of light intensity. Instead lights are automatically upgraded to be twice as bright. This creates more consistency and simplicity in light rigs. For example a directional light shining on a white diffuse surface will get the exact color of the light. The upgrade does not affect animation, thus if you have an animated light intensity value you must change your **animation curves** or script code and make them 2x as large to get the same look.

In the case of custom shaders where you define your own lighting functions, you need to remove the * 2 yourself.

```
// A common pattern in shader code that has this problem will look like this
c.rgb = s.Albedo * _LightColor0.rgb * (diff * atten * 2);
// You need to fix the code so it looks more like this
c.rgb = s.Albedo * _LightColor0.rgb * (diff * atten);
```

Increased interpolator and instruction counts for some surface shaders

Built-in lighting pipeline in Unity 5 can in some cases use more texture coordinate interpolators or math instruction count (to get things like non-uniform mesh scale, dynamic GI etc. working). Some of your existing **surface shaders** might be running into texture coordinate or ALU instruction limits, especially if they were targeting shader model 2.0 (default). Adding "#pragma target 3.0" can work around this issue. See http://docs.unity3d.com/Manual/SL-ShaderPrograms.html for the reference.

Non-uniform mesh scale has to be taken into account in shaders

In Unity 5.0, non-uniform meshes are not "prescaled" on the CPU anymore. This means that normal & tangent vectors can be non-normalized in the **vertex shader**. If you're doing manual lighting calculations there, you'd have to normalize them. If you're using Unity's surface shaders, then all necessary code will be generated for you.

Fog handling was changed

Unity 5.0 makes built-in Fog work on Windows Phone and consoles, but in order to achieve that we've changed how Fog is done a bit. For surface shaders and fixed function shaders, nothing extra needs to be done - fog will be added automatically (you can add "nofog" to surface shader #pragma line to explicitly make it not support fog).

For manually written vertex/**fragment shaders**, fog does not happen automagically now. You need to add #pragma multi_compile_fog and fog handling macros to your shader code. Check out built-in shader source, for example Unlit-Normal how to do it.

Surface shaders alpha channel change

By default all opaque surface shaders output 1.0 ("white") into alpha channel now. If you want to stop that, use "keepalpha" option on the #pragma surface line.

All alpha blended surface shaders use alpha **component** computed by the lighting function as blend factor now (instead of s.Alpha). If you're using custom lighting functions, you probably want to add something like "c.a = s.Alpha" towards the end of it.

Sorting by material index has been removed

Unity no longer sorts by material index in the forward renderloop. This improves performance because more objects can be rendered without state changes between them. This breaks compatibility for content that relies on material index as a way of sorting. In 4.x a **mesh** with two materials would always render the first material first, and the second material second. In Unity 5 this is not the case, the order depends on what reduces the most state changes to render the **scene**.

Fixed function TexGen, texture matrices and some SetTexture combiner modes were removed

Unity 5.0 removed support for this fixed function shader functionality:

UV coordinate generation (TexGen command).
UV transformation matrices (Matrix command on a texture property or inside a SetTexture).
Rarely used SetTexture combiner modes: signed add (a+-b), multiply signed add (a*b+-c), multiply subtract (a*b-c), dot product (dot3, dot3rgba).
Any of the above will do nothing now, and shader **inspector** will show warnings about their usage. You should rewrite affected shaders using programmable vertex+fragment shaders instead. All platforms support them nowadays, and there are no advantages whatsoever to use fixed function shaders.

If you have fairly old versions of Projector or Water shader packages in your project, the shaders there might be using this functionality. Upgrade the packages to 5.0 version.

Mixing programmable & fixed function shader parts is not allowed anymore

Mixing partially fixed function & partially programmable shaders (e.g. fixed function vertex lighting & pixel shader; or a vertex shader and texture combiners) is not supported anymore. It was never working on mobile, consoles or DirectX 11 anyway. This required changing behavior of Legacy/Reflective/VertexLit shader to not do that - it lost per-vertex specular support; on the plus side it now behaves consistently between platforms.

D3D9 shader compiler was switched from Cg 2.2 to HLSL

Mostly this should be transparent (just result in less codegen bugs and slightly faster shaders). However HLSL compiler can be slightly more picky about syntax. Some examples:

You need to fully initialize output variables. Use UNITY_INITIALIZE_OUTPUT helper macro, just like you would on D3D11.
“float3(a_4_component_value)” constructors do not work. Use “a_4_component_value.xyz” instead.

“unity_Scale” shader variable has been removed

The “unity_Scale” shader property has been removed. In 4.x unity_Scale.w was the 1 / uniform Scale of the transform, Unity 4.x only rendered non-scaled or uniformly scaled models. Other scales were performed on the CPU, which was very expensive & had an unexpected memory overhead.

In Unity 5.0 all this is done on the GPU by simply passing matrices with non-uniform scale to the shaders. Thus unity_Scale has been removed because it can not represent the full scale. In most cases where “unity_Scale” was used we recommend instead transforming to world space first. In the case of transforming normals, you always have to use normalize on the transformed normal now. In some cases this leads to slightly more expensive code in the vertex shader.

```
// Unity 4.x
float3 norm = mul ((float3x3)UNITY_MATRIX_IT_MV, v.normal * unity_Scale.w);
// Becomes this in Unity 5.0
float3 norm = normalize(mul ((float3x3)UNITY_MATRIX_IT_MV, v.normal));
```

```
// Unity 4.x
temp.xyzw = v.vertex.xzxz * unity_Scale.xzxz * _WaveScale4 + _WaveOffset;

// Becomes this in Unity 5.0
float4 wpos = mul (_Object2World, v.vertex);
temp.xyzw = wpos.xzxz * _WaveScale4 + _WaveOffset;
```

Shadows, Depth Textures and ShadowCollector passes

Forward rendered directional light shadows do not do separate “shadow collector” pass anymore. Now they calculate screenspace shadows from a **camera**’s depth texture (just like in deferred lighting).

This means that LightMode=ShadowCollector passes in shaders aren’t used for anything; you can just remove them from your shaders.

Depth texture itself is not generated using shader replacement anymore; it is rendered with ShadowCaster shader passes. This means that as long as your objects can cast proper shadows, then they will also appear in camera’s depth texture properly (was very hard to do before, if you wanted custom vertex animation or funky alpha testing). It also means that Camera-DepthTexture.shader is not used for anything now. And also, all built-in shadow shaders used no backface culling; that was changed to match culling mode of regular **rendering**.

Other Upgrade Notes for Unity 5.0

Leave feedback

More information about features that have changed and may affect your project when upgrading from Unity 4 to Unity 5

Locking / Hiding the Cursor

Cursor lock and cursor visibility are now independent of each other.

```
// Unity 4.x
Screen.lockCursor = true;
```

```
// Becomes this in Unity 5.0
Cursor.visible = false;
Cursor.lockState = CursorLockMode.Locked;
```

Linux

Gamepad handling has been reworked for Unity 5.

Unity is now capable of “configuring” gamepads - either via a database of known models, or using the SDL_GAMECONTROLLERCONFIG environment variable, which is set by Steam Big Picture / SteamOS for gamepads detected or configured with its interface.

Configured gamepads present a consistent layout: left stick uses axes 0/1, right stick uses axes 3/4, buttons on the face of the gamepad are 0–3, **etc**. To determine whether a gamepad has been configured, call Input.IsJoystickPreconfigured().

Windows Store Apps

‘Metro’ keyword was replaced to ‘WSA’ in most APIs, for example: BuildTarget.MetroPlayer became BuildTarget.WSAPlayer, PlayerSettings.Metro became PlayerSettings.WSA.

Defines in **scripts** like UNITY_METRO, UNITY_METRO_8_0, UNITY_METRO_8_1 are still there, but along with them there will be newer defines UNITY_WSA, UNITY_WSA_8_0, UNITY_WSA_8_1.

Other script API changes that cannot be upgraded automatically

UnityEngine.AnimationEvent is now a struct. Comparisons to ‘null’ will result in compile errors.

AddComponent(string), when called with a variable cannot be automatically updated to the generic version AddComponent<T>(). In such cases the API Updater will replace the call with a call to APIUpdaterRuntimeServices.AddComponent(). This method is meant to allow you to test your game in editor mode (they do a best effort to try to resolve the type at runtime) but it is not meant to be used in production, so it is an error to build a game with calls to such method). On platforms that support Type.GetType(string) you can try to use GetComponent(Type.GetType(typeName)) as a workaround.

AssetBundle.Load, AssetBundle.LoadAsync and AssetBundle.LoadAll have been deprecated. Use AssetBundle.LoadAsset, AssetBundle.LoadAssetAsync and AssetBundle.LoadAllAssets instead. Script updater cannot update them as the loading behaviors have changed a little. In 5.0 all the loading APIs will not load components any more, please use the new loading APIs to load the game object first, then look up the **component** on the object.

Unity Package changes

The internal package format of .unityPackages has changed, along with some behaviour of how packages are imported into unity and how conflicts are resolved.

Packages are now only constructed with the source **asset** and the .meta text file that contains all the importer settings for the **asset**.

Packages will always require asset importing now.

Packages will be reduced in size significantly, because the imported data (for example texture and audio data) will not be doubled up.

Packages with file names that already exist in the project, but with different GUIDs will have these files imported with a unique file name. This is done to prevent overriding files in the project that actually came from different packages or were created by the user.

Upgrading to Unity 4.0

Leave feedback

GameObject active state

Unity 4.0 changes how the active state of **GameObjects** is handled. GameObject's active state is now inherited by child **GameObjects**, so that any GameObject which is inactive will also cause its children to be inactive. We believe that the new behavior makes more sense than the old one, and should have always been this way. Also, the upcoming new GUI system heavily depends on the new 4.0 behavior, and would not be possible without it. Unfortunately, this may require some work to fix existing projects to work with the new Unity 4.0 behavior, and here is the change:

The old behavior:

Whether a GameObject is active or not was defined by its **.active** property.
This could be queried and set by checking the **.active** property.
A GameObject's active state had no impact on the active state of child GameObjects. If you want to activate or deactivate a GameObject and all of its children, you needed to call **GameObject.SetActiveRecursively**.
When using **SetActiveRecursively** on a GameObject, the previous active state of any child GameObject would be lost. When you deactivate and then activated a GameObject and all its children using **SetActiveRecursively**, any child which had been inactive before the call to **SetActiveRecursively**, would become active, and you had to manually keep track of the active state of children if you want to restore it to the way it was.
Prefabs could not contain any active state, and were always active after **prefab** instantiation.

The new behavior:

Whether a GameObject is active or not is defined by its own **.activeSelf** property, and that of all of its parents. The GameObject is active if its own **.activeSelf** property and that of all of its parents is **true**. If any of them are **false**, the GameObject is inactive.
This can be queried using the **.activeInHierarchy** property.
The **.activeSelf** state of a GameObject can be changed by calling **GameObject.SetActive**. When calling **SetActive (false)** on a previously active GameObject, this will deactivate the GameObject and all its children. When calling **SetActive (true)** on a previously inactive GameObject, this will activate the GameObject, if all its parents are active. Children will be activated when all their parents are active (i.e., when all their parents have **.activeSelf** set to **true**).
This means that **SetActiveRecursively** is no longer needed, as active state is inherited from the parents. It also means that, when deactivating and activating part of a hierarchy by calling **SetActive**, the previous active state of any child GameObject will be preserved.
Prefabs can contain active state, which is preserved on prefab instantiation.

Example:

You have three GameObjects, A, B and C, so that B and C are children of A.

Deactivate C by calling **C.SetActive(false)**.
Now, **A.activeInHierarchy == true**, **B.activeInHierarchy == true** and **C.activeInHierarchy == false**.
Likewise, **A.activeSelf == true**, **B.activeSelf == true** and **C.activeSelf == false**.
Now we deactivate the parent A by calling **A.SetActive(false)**.

Now, **A.activeInHierarchy == false**, **B.activeInHierarchy == false** and **C.activeInHierarchy == false**.
Likewise, **A.activeSelf == false**, **B.activeSelf == true** and **C.activeSelf == false**.
Now we activate the parent A again by calling **A.SetActive(true)**.
Now, we are back to **A.activeInHierarchy == true**, **B.activeInHierarchy == true** and **C.activeInHierarchy == false**.
Likewise, **A.activeSelf == true**, **B.activeSelf == true** and **C.activeSelf == false**.

The new active state in the editor

To visualize these changes, in the Unity 4.0 editor, any GameObject which is inactive (either because it's own **.activeSelf** property is set to **false**, or that of one of it's parents), will be greyed out in the hierarchy, and have a greyed out icon in the **inspector**. The GameObject's own **.activeSelf** property is reflected by it's active checkbox, which can be toggled regardless of parent state (but it will only activate the GameObject if all parents are active).

How this affects existing projects:

To make you aware of places in your code where this might affect you, the **GameObject.active** property and the **GameObject.SetActiveRecursively()** function have been deprecated.
They are, however still functional. Reading the value of **GameObject.active** is equivalent to reading **GameObject.activeInHierarchy**, and setting **GameObject.active** is equivalent to calling **GameObject.SetActive()**. Calling **GameObject.SetActiveRecursively()** is equivalent to calling **GameObject.SetActive()** on the GameObject and all of it's children.
Exiting **scenes** from 3.5 are imported by setting the **selfActive** property of any GameObject in the scene to it's previous **active** property.
As a result, any project imported from previous versions of Unity should still work as expected (with compiler warnings, though), as long as it does not rely on having active children of inactive GameObjects (which is no longer possible in Unity 4.0).
If your project relies on having active children of inactive GameObjects, you need to change your logic to a model which works in Unity 4.0.

Changes to the asset processing pipeline

During the development of 4.0 our asset import pipeline has changed in some significant ways internal in order to improve performance, memory usage and determinism. For the most part these changes does not have an impact on the user with one exception: Objects in assets are not made persistent until the very end of the import pipeline and any previously imported version of an assets will be completely replaced.

The first part means that during post processing you cannot get the correct references to objects in the asset and the second part means that if you use the references to a previously imported version of the asset during post processing do store modification those modifications will be lost.

Example of references being lost because they are not persistent yet

Consider this small example:

```
public class ModelPostprocessor : AssetPostprocessor
{
    public void OnPostprocessModel(GameObject go)
```

```
    {
        PrefabUtility.CreatePrefab("Prefabs/" + go.name, go);
    }
}
```

In Unity 3.5 this would create a prefab with all the correct references to the meshes and so on because all the meshes would already have been made persistent, but since this is not the case in Unity 4.0 the same post processor will create a prefab where all the references to the meshes are gone, simply because Unity 4.0 does not yet know how to resolve the references to objects in the original model prefab. To correctly copy a modelprefab in to prefab you should use **OnPostProcessAllAssets** to go through all imported assets, find the modelprefab and create new prefabs as above.

Example of references to previously imported assets being discarded

The second example is a little more complex but is actually a use case we have seen in 3.5 that broke in 4.0. Here is a simple **ScriptableObject** with a references to a **mesh**.

```
public class Referencer : ScriptableObject
{
    public Mesh myMesh;
}
```

We use this **ScriptableObject** to create an asset with references to a mesh inside a model, then in our post processor we take that reference and give it a different name, the end result being that when we have reimported the model the name of the mesh will be what the post processor determines.

```
public class Postprocess : AssetPostprocessor
{
    public void OnPostprocessModel(GameObject go)
    {
        Referencer myRef = (Referencer)AssetDatabase.LoadAssetAtPath("Assets/MyR
        myRef.myMesh.name = "AwesomeMesh";
    }
}
```

This worked fine in Unity 3.5 but in Unity 4.0 the already imported model will be completely replaced, so changing the name of the mesh from a previous import will have no effect. The Solution here is to find the mesh by some other means and change its name. What is most important to note is that in Unity 4.0 you should ONLY modify the given input to the post processor and not rely on the previously imported version of the same asset.

Mesh Read/Write option

Unity 4.0 adds a “Read/Write Enabled” option in the Model import settings. When this option is turned off, it saves memory since Unity can unload a copy of mesh data in the game.

However, if you are scaling or instantiating meshes at runtime with a non-uniform scale, you may have to enable “Read/Write Enabled” in their import settings. The reason is that non-uniform scaling requires the mesh data to be kept in memory. Normally we detect this at build time, but when meshes are scaled or instantiated at runtime you need to set this manually. Otherwise they might not be rendered in game builds correctly.

Mesh optimization

The Model Importer in Unity 4.0 has become better at mesh optimization. The “Mesh Optimization” checkbox in the Model Importer in Unity 4.0 is now enabled by default, and will reorder the vertices in your Mesh for optimal performance. You may have some post-processing code or effects in your project which depend on the vertex order of your meshes, and these might be broken by this change. In that case, turn off “Mesh Optimization” in the Mesh importer. Especially, if you are using the SkinnedCloth **component**, mesh optimization will cause your vertex weight mapping to change. So if you are using SkinnedCloth in a project imported from 3.5, you need to turn off “Mesh Optimization” for the affected meshes, or reconfigure your vertex weights to match the new vertex order.

Mobile input

With Unity 4.0 mobile sensor input got better alignment between platforms, which means you can write less code when handling typical input on mobile platforms. Now acceleration and gyro input will follow screen orientation in the same way both on **iOS** and Android platforms. To take advantage of this change you should refactor your input code and remove platform and screen orientation specific code when handling acceleration and gyro input. You still can get old behavior on **iOS** by setting **Input.compensateSensors** to false.

Upgrading to Unity 3.5

Leave feedback

If you have an FBX file with a **root node** marked up as a skeleton, it will be imported with an additional **root node** in 3.5, compared to 3.4

Unity 3.5 does this because when importing animated characters, the most common setup is to have one root node with all bones below and a skeleton next to it in the hierarchy. When creating additional animations, it is common to remove the skinned **mesh** from the fbx file. In that case the new import method ensures that the additional root node always exists and thus animations and the skinned **mesh** actually match.

If the connection between the instance and the FBX file's **prefab** has been broken in 3.4 the animation will not match in 3.5, and as a result your animation might not play.

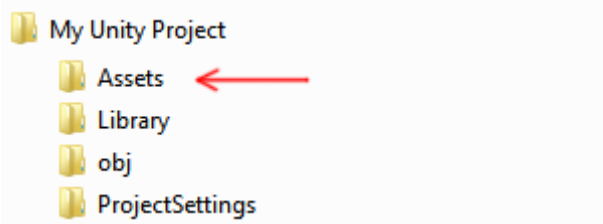
In that case it is recommended that you recreate the prefabs or Game Object hierarchies by dragging your FBX file into your **scene** and recreating it.

Importing

Leave feedback

You can bring **Assets** created outside of Unity into your Unity Project by either exporting the file directly into the **Assets** folder under your Project, or copying it into that folder. For many common formats, you can save your source file directly into your Project's **Assets** folder and Unity can read it. Unity also detects when you save new changes to the file and re-imports files as necessary.

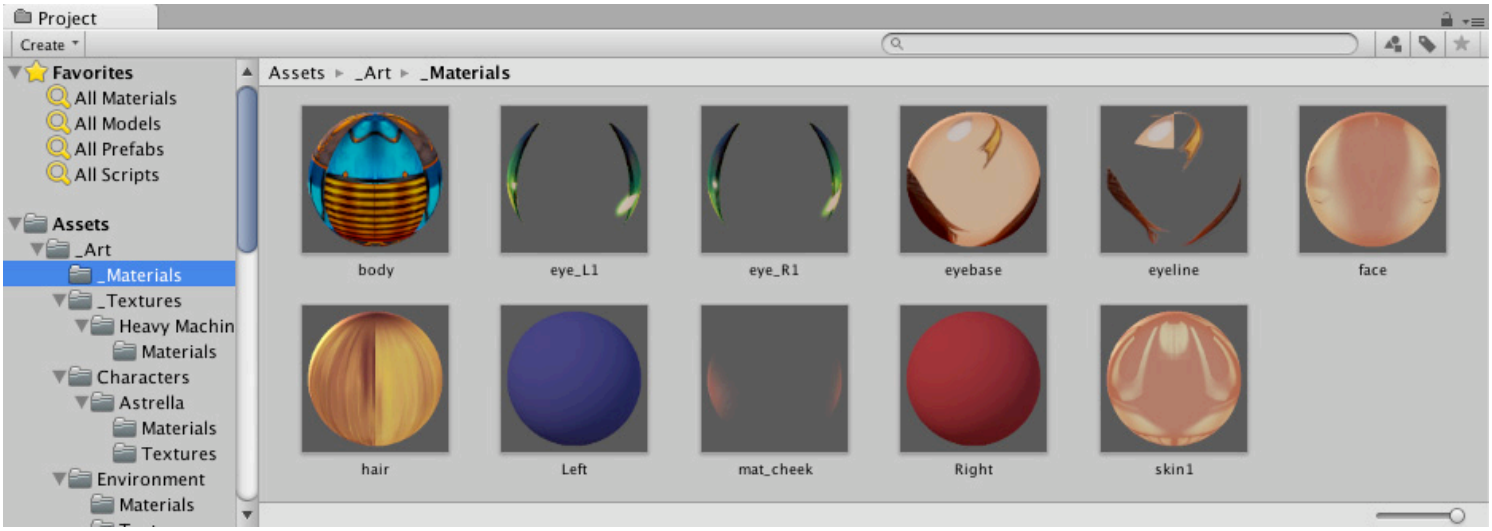
When you create a Unity Project, you are creating a folder (named after your Project) which contains the following subfolders:



The basic file structure of a Unity Project

Save or copy files that you want to use in your Project into the Assets folder. You can use the **Project window** inside Unity to view the contents of your Assets folder. Therefore, if you save or copy a file to your Assets folder, Unity imports it and appears in your Project window.

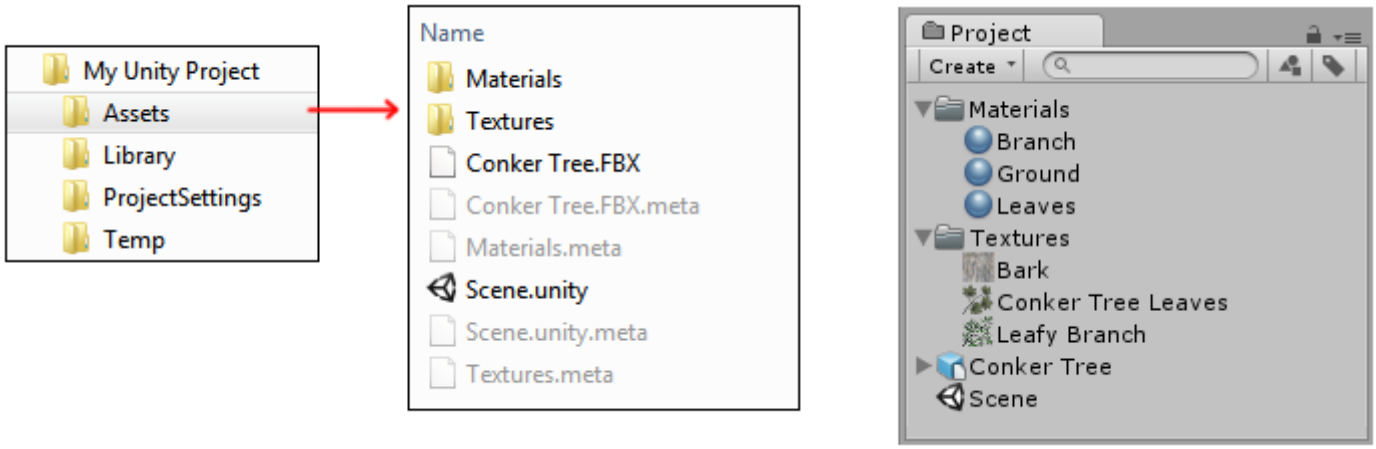
Unity automatically detects files as they are added to the Assets folder, or if they are modified. When you put any Asset into your Assets folder, it appears in your **Project View**.



The Project Window shows Assets that Unity imported into your Project

If you drag a file into Unity's Project window from your computer (either from the Finder on Mac, or from the Explorer on Windows), Unity *copies* it into your Assets folder, and it appears in the Project window.

The items that appear in your Project window represent (in most cases) actual files on your computer, and if you delete them within Unity, you are deleting them from your computer too.



The relationship between the Assets Folder in your Unity Project on your computer, and the Project window within Unity

The above image shows an example of a few files and folders inside the Assets folder of a Unity Project. You can create as many folders as you like and use them to organize your Assets.

The image above shows **.meta** files listed in the file system, but not visible in Unity's Project window. Unity creates these .meta files for each asset and folder, but they are hidden by default, so you may not see them in your file system either.

They contain important information about how the asset is used in the Project and they must stay with the asset file they relate to, so if you move or rename an asset file in your file system, you must also move/rename the meta file to match.

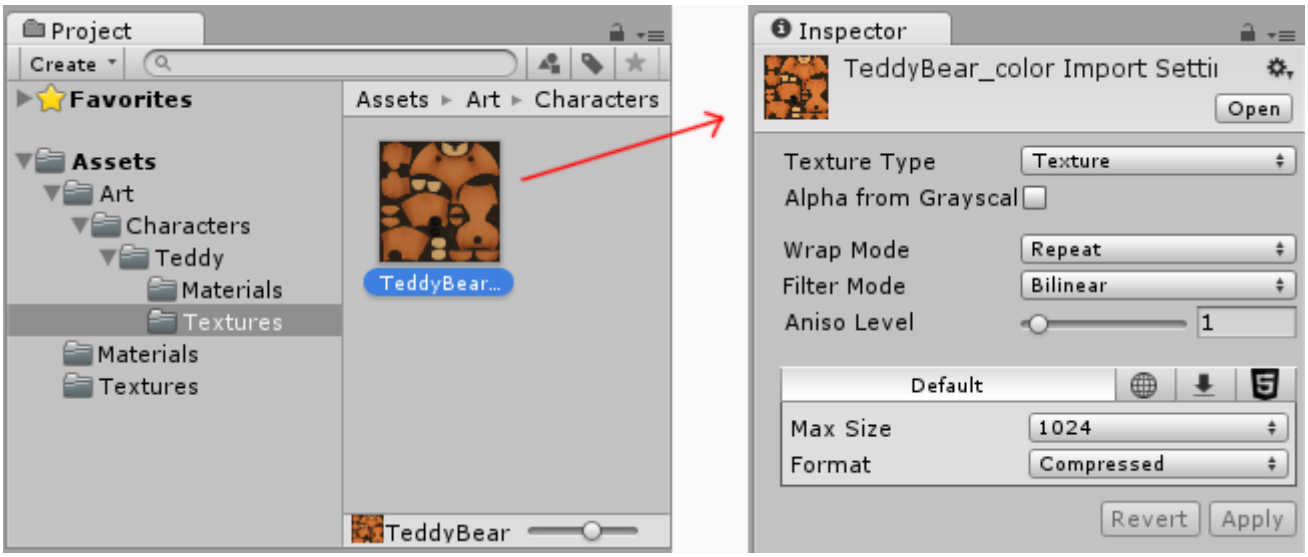
The simplest way to safely move or rename your assets is to always do it from within Unity's Project folder. This way, Unity automatically moves or renames the corresponding meta file. If you like, you can read more about .meta files and what goes on behind-the-scenes during the import process.

If you want to bring collections of Assets into your Project, you can use Asset packages.

Inspecting Assets

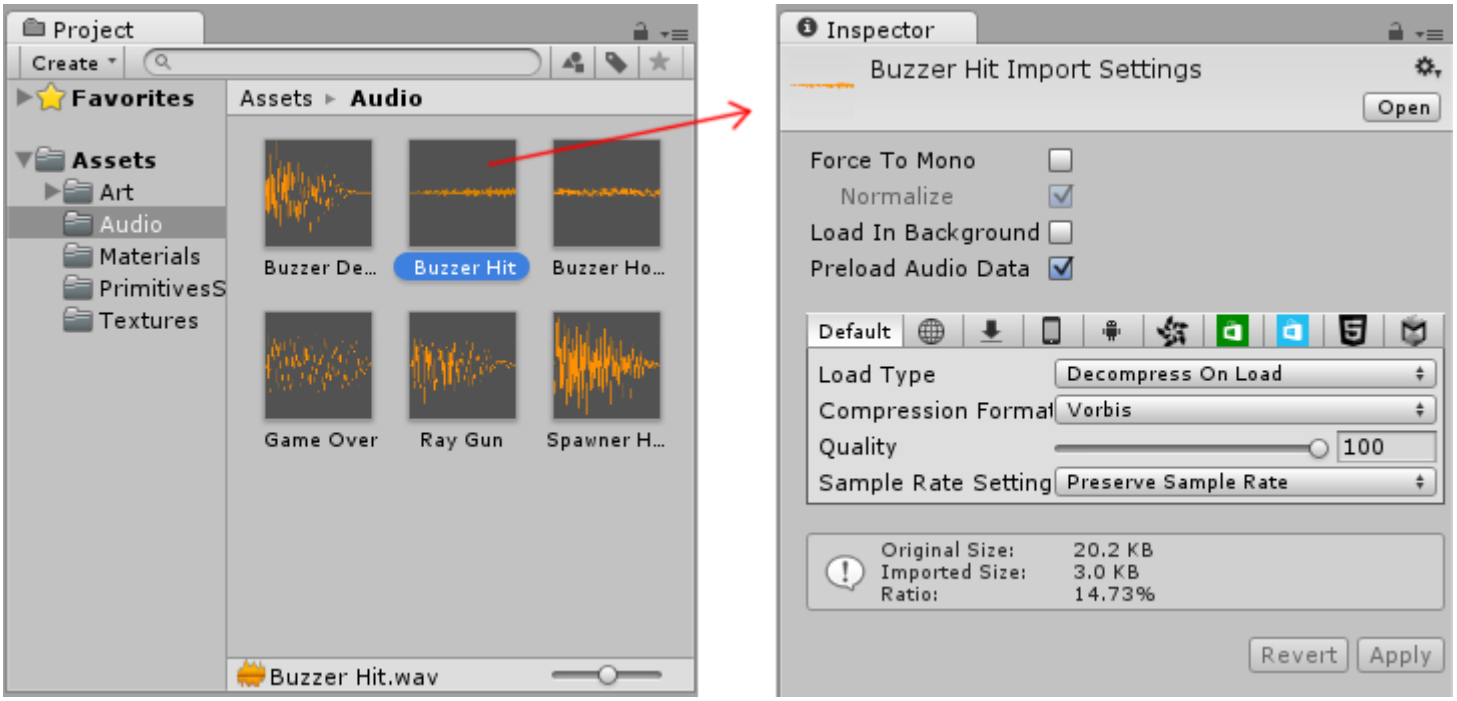
Each type of Asset that Unity supports has a set of Import Settings, which affect how the Asset appears or behaves. To view an Asset's import settings, select the Asset in the **Project View**. The import settings for this Asset will appear in the **Inspector**. The options that appear vary depending on the type of Asset selected.

For example, the import settings for an image allow you to choose whether Unity imports it as a Texture, a 2D **sprite**, or a normal map. The import settings for an FBX file allow you to adjust the scale, generate normals or **lightmap** coordinates, and split & trim **animation clips** defined in the file.



Clicking on an image Asset in the Project Window shows the import settings for that Asset in the Inspector

For other Asset types, the import settings look different. The various settings you see relate to the type of Asset selected. Here's an example of an Audio Asset, with its related import settings shown in the inspector:



An Audio Asset selected in the Project Window shows the Audio import settings for that Asset in the Inspector

If you are developing a cross-platform Project, you can override the "default" settings and assign different import settings on a per-platform basis.

See also:

Packages
Materials and Shaders
Textures and Videos
Sprite Editor
Sprite Packer

Audio Files
Tracker Modules

2018–04–25 Page amended with limited editorial review

Model import workflows

Leave feedback

Note: This workflow assumes you already have a **Model file** to import. If you don't have a file already, you can read the guidelines on how to export an FBX file before exporting it from your 3D modeling software. For guidelines on how to export *Humanoid* animation from your 3D modeling software, see Humanoid Asset preparation.

Model files can contain a variety of data, such as character Meshes, Animation Rigs and Clips, as well as Materials and Textures. Most likely, your file does not contain all of these elements at once, but you can follow any portion of the workflow that you need to.

Accessing the Import Settings window

No matter what kind of data you want to extract from the Model file, you always start the same way: 1. Open the **Project** window and the **Inspector** so that you can see both at once. 2. Select the **Model** file you want to import from the **Asset** folder in the **Project** window. The **Import Settings** window opens in the **Inspector** showing the **Model tab** by default.

Setting Model-specific and general importer options

Then you can set any of these options on the Model tab:

Use the **Scale Factor** and **Convert Units** properties to adjust how Unity interprets units. For example, 3ds Max uses 1 unit to represent 10 centimeters, whereas Unity uses 1 unit to represent 1 meter.
Use the **Mesh Compression**, **Read/Write Enabled**, **Optimize Mesh**, **Keep Quads**, **Index Format**, or **Weld Vertices** properties to reduce resources and save memory.
You can enable the **Import BlendShapes** option if the Model file came from Maya or 3ds Max, or any other 3d modeling application that supports morph target animation.
You can enable the **Generate Colliders** option for environment geometry.
You can enable specific FBX settings, such as **Import Visibility**, or **Import Cameras** and **Import Lights**.
For Model files that contain only Animation, you can enable the **Preserve Hierarchy** option to prevent hierarchy mismatch in your skeleton.
You can set the **Swap UVs** and **Generate Lightmap UVs** if you are using a **Lightmap**.
You can exercise control over how Unity handles the Normals and Tangents in your Model with the **Normals**, **Normals Mode**, **Tangents**, or **Smoothing Angle** options.

Setting options for importing Rigs and Animation

If your file contains Animation data, you can follow the guidelines for setting up the Rig using the Rig tab and then extracting or defining **Animation Clips** using the Animation tab. The workflow differs between Humanoid and Generic (non-Humanoid) animation types because Unity needs the Humanoid's bone structure to be very specific, but only needs to know which bone is the **root node** for the Generic type:

Humanoid-type workflow
Generic-type workflow

Dealing with Materials and Textures

If your file contains **Material** or **Texture**, you can define how you want to deal with them:

Click the Materials tab in the **Import Settings** window.

Enable the **Import Materials** option. Several options appear in the **Materials** tab, including the **Location** option, whose value determines what the other options are.
Choose the **Use Embedded Materials** option to keep the imported Materials inside the imported Asset.
When you have finished setting the options, click the **Apply** button at the bottom of the **Import Settings** window to save them or click the **Revert** button to cancel.

Finally, you can import the file into your scene:

If it contains a **Mesh**, drag the file into the **Scene** view to instantiate it as a **Prefab** for a **GameObject**.
If it contains Animation Clips, you can drag the file into the Animator window to use in the State Machine or onto an **Animation track** on the Timeline window. You can also drag the Animation take directly onto an instantiated Prefab in the **Scene view**. This automatically creates an Animation Controller and connects the animation to the Model.

2018–04–25 Page amended with limited editorial review

2018–09–26 Page amended with limited editorial review

Importing humanoid animations

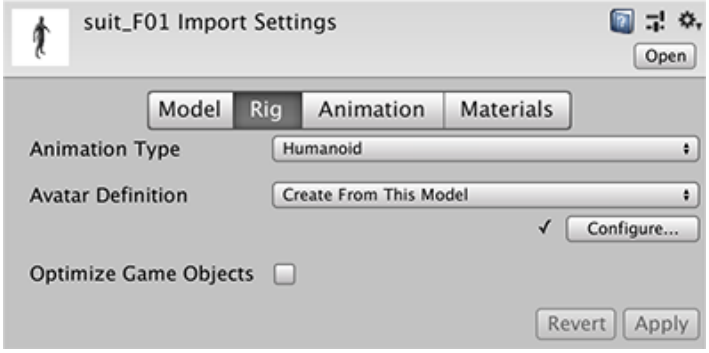
Leave feedback

When Unity imports **Model files** that contain **Humanoid** Rigs and Animation, it needs to reconcile the bone structure of the Model to its Animation. It does this by mapping each bone in the file to a Humanoid Avatar so that it can play the Animation properly. For this reason, it is important to carefully prepare your Model file before importing into Unity.

Define the Rig type and create the Avatar.
Correct or verify the Avatar's mapping.
Once you are finished with the bone mapping, you can optionally click the **Muscles & Settings** tab to tweak the Avatar's muscle configuration.
You can optionally save the mapping of your skeleton's bones to the Avatar as a **Human Template** (.ht) file.
You can optionally limit the animation that gets imported on certain bones by defining an Avatar Mask.
From the **Animation** tab, enable the **Import Animation** option and then set the other Asset-specific properties,
.
If the file consists of multiple animations or actions, you can define specific action ranges as Animation Clips.
For each **Animation Clip** defined in the file, you can:
Change the pose and root transform
Optimize looping
Mirror the animation on both sides of the Humanoid skeleton.
Add curves to the clip in order to animate the timings of other items
Add events to the clip in order to trigger certain actions in time with the animation
Discard part of the animation similar to using a runtime Avatar Mask but applied at import time
Select a different Root Motion Node to drive the action from
Read any messages from Unity about importing the clip
Watch a preview of the animation clip
To save your changes, click the **Apply** button at the bottom of the **Import Settings** window or **Revert** to discard your changes.

Set up the Avatar

From the Rig tab of the Inspector window, set the **Animation Type** to **Humanoid**. By default, the **Avatar Definition** property is set to **Create From This Model**. If you keep this option, Unity attempts to map the set of bones defined in the file to a Humanoid Avatar.



Humanoid Rig

Note: In some cases, you can change this option to **Copy From Other Avatar** to use an Avatar you already defined for another Model file. For example, if you create a **Mesh** (skin) in your 3D modeling application with several distinct animations, you can export the Mesh to one FBX file, and each animation to its own FBX file. When you import these files into Unity, you only need to create a single Avatar for the first file you import (usually the Mesh). As long as all the files use the same bone structure, you can re-use that Avatar for the rest of the files (for example, all the animations).

If you enable this option, you must specify which Avatar you want to use by setting the **Source** property.

Click the **Apply** button. Unity tries to match up the existing bone structure to the Avatar bone structure. In many cases, it can do this automatically by analyzing the connections between bones in the rig.

If the match succeeds, a check mark appears next to the **Configure** menu. Unity also adds an Avatar sub-asset to the Model Asset, which you can find in the **Project view**.

Avatar appears as a sub-asset of the imported Model

A successful match simply means that Unity was able to match all of the required bones. However, for better results, you also need to match the optional bones and set the model in a proper **T-pose**.

If Unity can't create the Avatar, a cross appears next to the **Configure** button, and no Avatar sub-asset appears in the Project view.

Unity failed to create a valid Avatar

Since the Avatar is such an important aspect of the animation system, it is important to configure it properly for your **Model**.

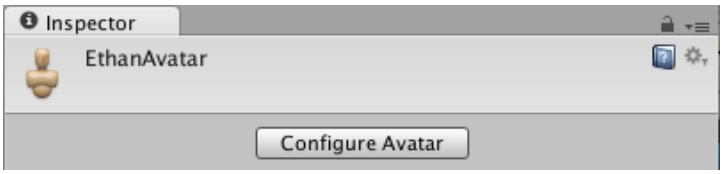
For this reason, whether or not the automatic Avatar creation succeeds, you should always check that your Avatar is valid and properly set up.

Configure the Avatar

If Unity failed to create the Avatar for your model, you must click the **Configure ...** button on the **Rig** tab to open the Avatar window and fix your Avatar.

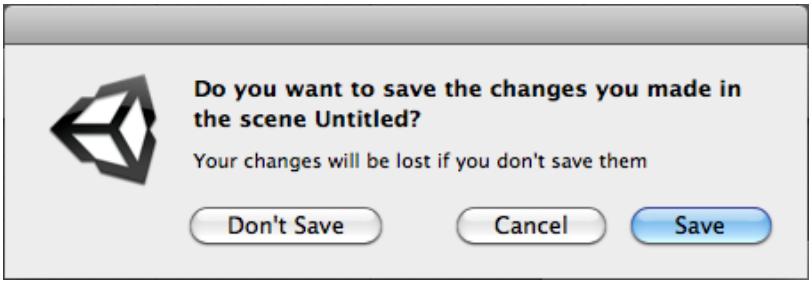
If the match was successful, you can either click the **Configure ...** button on the **Rig** tab or open the window from the Project view:

Click the Avatar sub-Asset in the **Project** view. The **Inspector** displays the name of the Avatar and a **Configure Avatar** button.
Click the **Configure Avatar** button.



The Inspector for an Avatar sub-asset

If you haven't already saved the Avatar, a message appears asking you to save your scene:



The reason for this is that in **Configure** mode, the **Scene view** is used to display bone, muscle and animation information for the selected model alone, without displaying the rest of the scene.

Once you have saved the scene, the **Avatar** window appears in the **Inspector** displaying any bone mapping.

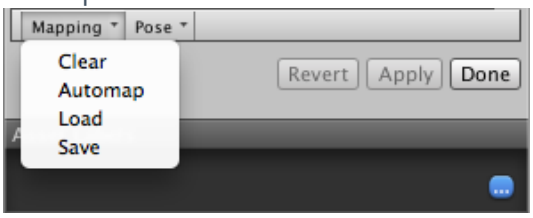
Make sure the bone mapping is correct and that you map any optional bones that Unity did not assign.

Your skeleton needs to have at least the required bones in place for Unity to produce a valid match. In order to improve your chances for finding a match to the Avatar, name your bones in a way that reflects the body parts they represent. For example, "LeftArm" and "RightForearm" make it clear what these bones control.

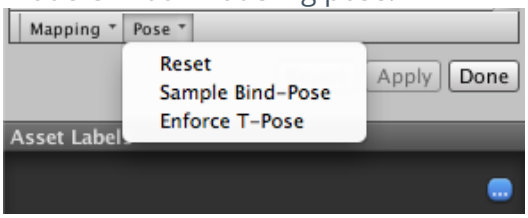
Mapping strategies

If the model does *not* yield a valid match, you can use a similar process to the one that Unity uses internally:

Choose **Clear** from the **Mapping** menu at the bottom of the **Avatar** window to reset any mapping that Unity attempted.



Choose **Sample Bind-pose__from the** Pose__ menu at the bottom of the **Avatar** window to approximate the Model's initial modeling pose.



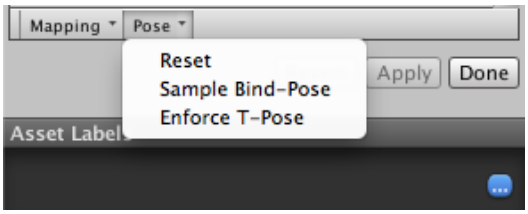
Choose **Mapping** > **Automap** to create a bone-mapping from an initial pose.
Choose **Pose** > **Enforce T-Pose** to set the Model back to to required T-pose.

If automapping fails completely or partially, you can manually assign bones by either dragging them from the **Scene** view or from the **Hierarchy** view. If Unity thinks a bone fits, it appears in green in the **Avatar Mapping** tab; otherwise it appears in red.

Resetting the pose

The **T-pose** is the default pose required by Unity animation and is the recommended pose to model in your 3D modeling application. However, if you did not use the T-pose to model your character and the animation does not work as expected, you

can choose the :



The **Pose** drop-down menu at the bottom of the **Avatar** window

If the bone assignment is correct, but the character is not in the correct *pose*, you will see the message "Character not in T-Pose". You can try to fix that by choosing **Enforce T-Pose** from the **Pose** menu. If the pose is still not correct, you can manually rotate the remaining bones into a T-pose.

Creating an Avatar Mask

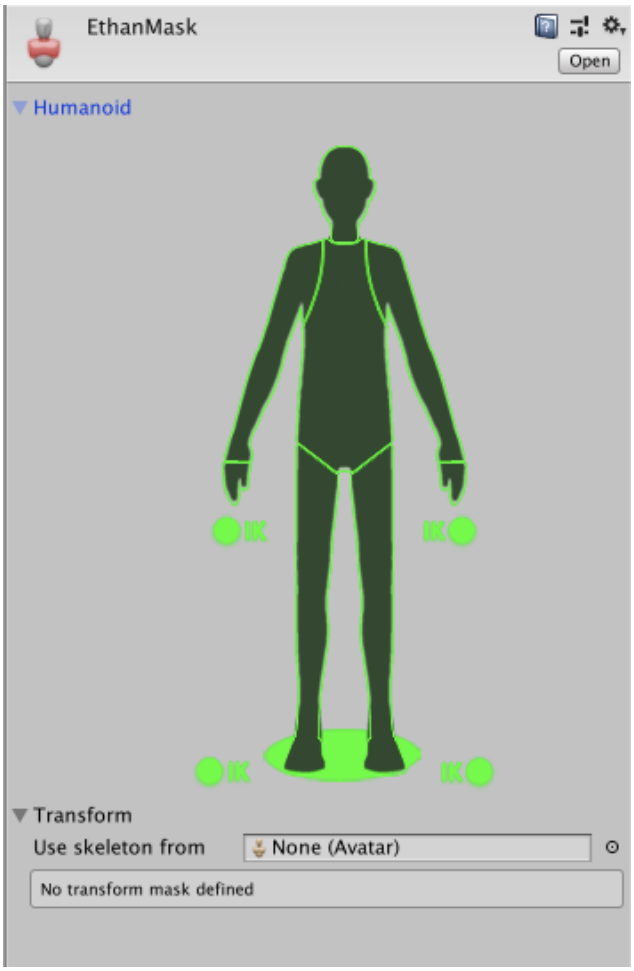
Masking allows you to discard some of the animation data within a clip, allowing the clip to animate only parts of the object or character rather than the entire thing. For example, you may have a standard walking animation that includes both arm and leg motion, but if a character is carrying a large object with both hands then you wouldn't want their arms to swing to the side as they walk. However, you could still use the standard walking animation while carrying the object by using a mask to only play the upper body portion of the carrying animation over the top of the walking animation.

You can apply masking to animation clips either during import time, or at runtime. Masking during import time is preferable, because it allows the discarded animation data to be omitted from your build, making the files smaller and therefore using less memory. It also makes for faster processing because there is less animation data to be blended at runtime. In some cases, import masking may not be suitable for your purposes. In that case, you can apply a mask at runtime by creating an **Avatar Mask** Asset, and using it in the layer settings of your **Animator Controller**.

To create an empty Avatar Mask Asset, you can either:

Choose **Create** > **Avatar Mask** from the **Assets** menu.
Click the Model object you want to define the mask on in the **Project** view, and then right-click and choose **Create** > **Avatar Mask**.

The new Asset appears in the **Project** view:



The Avatar Mask window

You can now add portions of the body to the mask and then add the mask to either an Animation Layer or add a reference to it under the Mask section of the Animation tab.

2018–04–25 Page amended with limited editorial review

Importing non-humanoid animations

Leave feedback

A *Humanoid* model is a very specific structure, containing at least 15 bones organized in a way that loosely conforms to an actual human skeleton. Everything else that uses the Unity Animation System falls under the *non-Humanoid*, or **Generic** category. This means that a Generic Model might be anything from a teakettle to a dragon, so non-Humanoid skeletons could have a huge range of possible bone structures.

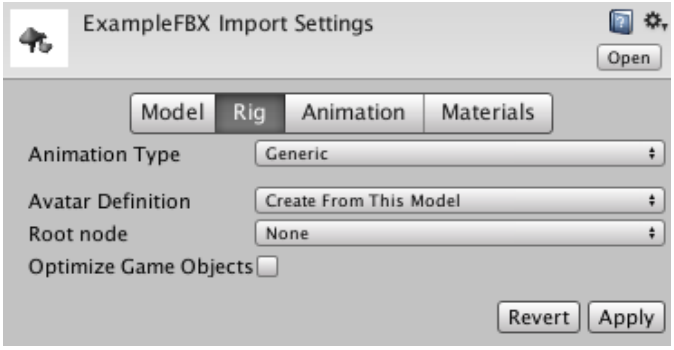
The solution to dealing with this complexity is that Unity only needs to know which bone is the **Root node**. In terms of the Generic character, this is the best approximation to the Humanoid character's **center of mass**. It helps Unity determine how to render animation in the most faithful way possible. Since there is only one bone to map, Generic setups do not use the Humanoid Avatar window. As a result, preparing to import your non-Humanoid **Model file** into Unity requires fewer steps than for Humanoid models.

Set up your Rig as **Generic**.
You can optionally limit the animation that gets imported on certain bones by defining an Avatar Mask.
From the **Animation** tab, enable the **Import Animation** option and then set the other Asset-specific properties, .
If the file consists of multiple animations or actions, you can define specific frame ranges as Animation Clips.
For each **Animation Clip** defined in the file, you can:
Set the pose and root transform
Optimize looping
Add curves to the clip in order to animate the timings of other items
Add events to the clip in order to trigger certain actions in time with the animation
Discard part of the animation similar to using a runtime Avatar Mask but applied at import time
Select a different Root Motion Node to drive the action from
Read any messages from Unity about importing the clip
Watch a preview of the animation clip
To save your changes, click the **Apply** button at the bottom of the **Import Settings** window or **Revert** to discard your changes.

Setting up the Rig

From the Rig tab of the Inspector window, set the **Avatar** (animation) type to **Generic**. By default, the **Avatar Definition** property is set to **Create From This Model** and the **Root node** option is set to **None**.

In some cases, you can change the **Avatar Definition** option to **Copy From Other Avatar** to use an Avatar you already defined for another Model file. For example, if you create a **Mesh** (skin) in your 3D modeling application with several distinct animations, you can export the Mesh to one FBX file, and each animation to its own FBX file. When you import these files into Unity, you only need to create a single Avatar for the first file you import (usually the Mesh). As long as all the files use the same bone structure, you can re-use that Avatar for the rest of the files (for example, all the animations).



Generic Rig

If you keep the **Create From This Model** option, you must then choose a bone from the **Root node** property.

If you decide to change the **Avatar Definition** option to **Copy From Other Avatar**, you need to specify which Avatar you want to use by setting the **Source** property.

Click the **Apply** button. Unity creates a **Generic Avatar** and adds an an Avatar sub-asset under the Model Asset, which you can find in the **Project view**.

Avatar appears as a sub-asset of the imported Model

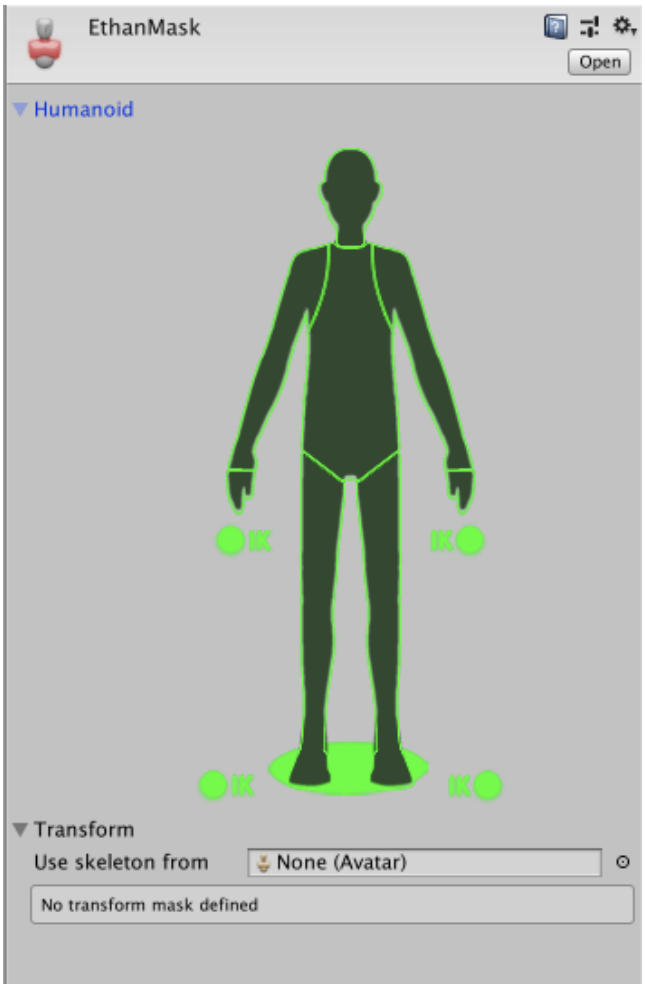
Note: The Generic Avatar is not the same thing as the Humanoid Avatar, but it does appear in the Project view, and it does hold the Root node mapping. However, if you click on the Avatar icon in the Project view to display its properties in the **Inspector**, only its name appears and there is no **Configure Avatar** button.

Creating an Avatar Mask

You can apply masking to animation clips either during import time, or at runtime. Masking during import time is preferable, because it allows the discarded animation data to be omitted from your build, making the files smaller and therefore using less memory. It also makes for faster processing because there is less animation data to be blended at runtime. In some cases, import masking may not be suitable for your purposes. In that case, you can apply a mask at runtime by creating an **Avatar Mask** Asset, and using it in the layer settings of your **Animator Controller**.

To create an empty Avatar Mask Asset, you can either:

Choose **Create** > **Avatar Mask** from the **Assets** menu.
Click the Model object you want to define the mask on in the **Project** view, and then right-click and choose **Create** > **Avatar Mask**.
The new Asset appears in the **Project** view:



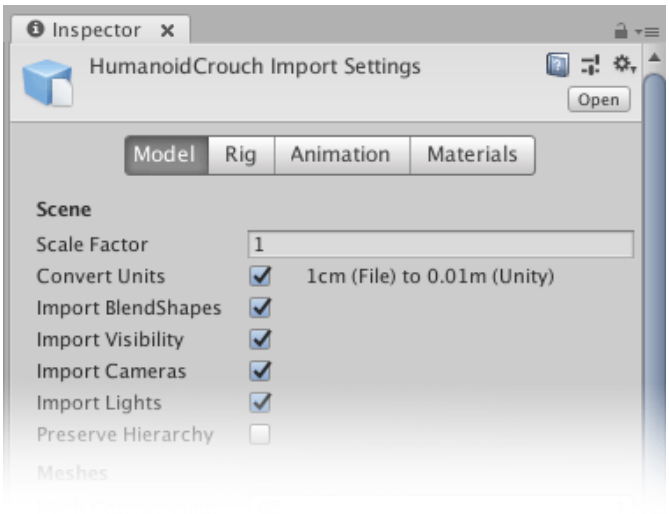
The Avatar Mask window

You can now choose which bones to include or exclude from a Transform hierarchy and then add the mask to either an Animation Layer or add a reference to it under the Mask section of the Animation tab.

2018–04–25 Page amended with limited editorial review

Model Import Settings window

Leave feedback



The Import Settings window

When you put **Model files** in the **Assets** folder under your Unity Project, Unity automatically imports and stores them as Unity **Assets**. To view the import settings in the **Inspector**, click on the file in the **Project** window. You can customize how Unity imports the selected file by setting the properties on four tabs on this window:

Model

A 3D Model can represent a character, a building, or a piece of furniture. In these cases, Unity creates multiple Assets from a single model file. In the Project window, the main imported object is a model **Prefab**. Usually there are also several **Mesh** objects that the model Prefab references.

Rig

A Rig (sometimes called a *skeleton*) comprises a set of deformers arranged in a hierarchy that animate a Mesh (sometimes called *skin*) on one or more models created in a 3D modeling application such as as Autodesk® 3ds Max® or Autodesk® Maya®. For **Humanoid** and **Generic** (non-humanoid) Models, Unity creates an **Avatar** to reconcile the imported Rig with the Unity **GameObject**.

Animation

You can define any series of different poses occurring over a set of frames, such as walking, running, or even idling (shifting from one foot to the other) as an Animation Clip. You can reuse clips for any Model that has an identical Rig. Often a single file contains several different actions, each of which you can define as a specific **Animation Clip**.

Materials

You can extract Materials and Textures or leave them embedded within the model. You can also adjust how Material is mapped in the Model.

See also

Model import workflows: Overview of importing Model files
Model file formats: Which file formats (both proprietary and generic) that Unity supports, as well as issues specific to various 3D modeling software applications
Exporting from other applications: General guidelines for how to export FBX (and proprietary) files from your 3D modeling applications

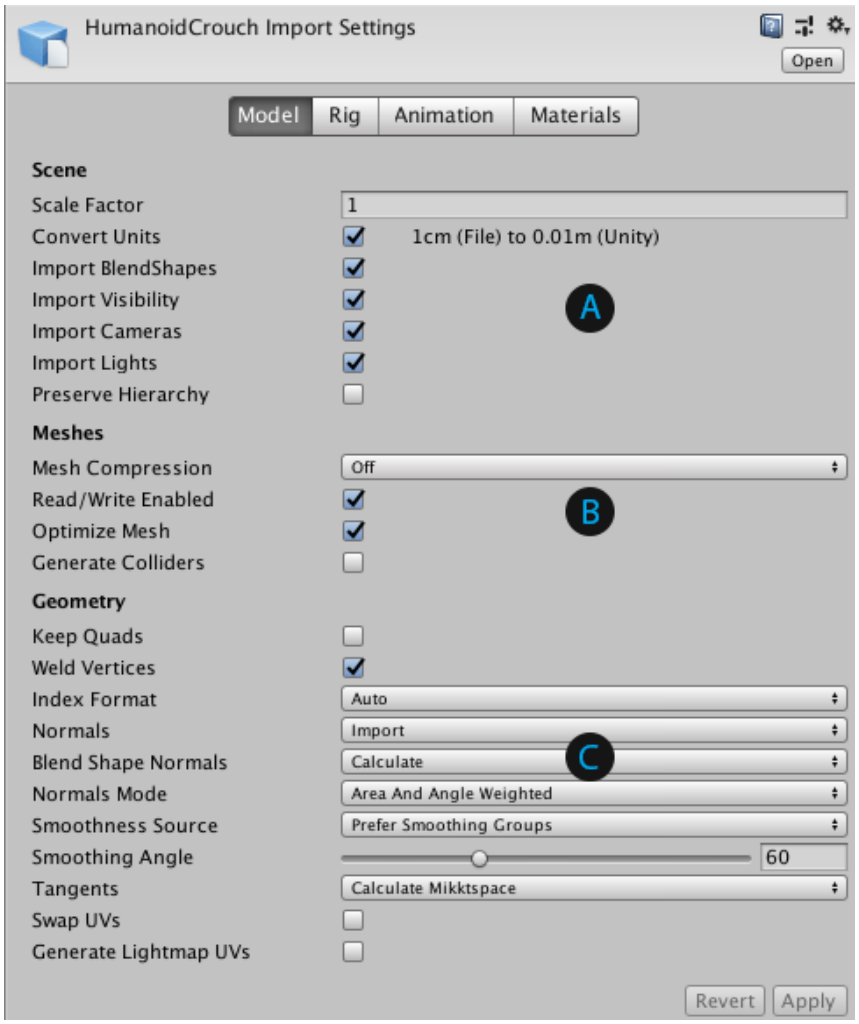
---

2018–04–25 Page amended with limited editorial review

Model tab

Leave feedback

The **Import Settings** for a **Model file** appear in the **Model** tab of the **Inspector** window when you select the Model. These settings affect various elements and properties stored inside the Model. Unity uses these settings to import each **Asset**, so you can adjust any settings to apply to different Assets in your Project.



Import settings for the Model

This section provides information about each of the sections on the Model tab:

A Scene-level properties, such as whether to import Lights and Cameras, and what scale factor to use.

B Properties specific to Meshes.

C Geometry-related properties, for dealing with topology, UVs, and normals.

Scene

Property	*Function*
Scale Factor	Set this value to apply a global scale on the imported Model whenever the original file scale (from the Model file) does not fit the intended scale in your Project. Unity's physics system expects 1 meter in the game world to be 1 unit in the imported file.
Convert Units	Enable this option to convert the Model scaling defined in the Model file to Unity's scale.

Property	*Function*
Import BlendShapes	Allow Unity to import BlendShapes with your Mesh. **Note** Importing blendshape normals requires smoothing groups in the FBX file.
Import Visibility	Import the FBX settings that define whether or not MeshRenderer **components** are enabled (visible). See Importing Visibility below for details.
Import Cameras	Import cameras from your .FBX file. See Importing Cameras below for details.
Import Lights	Import lights from your .FBX file. See Importing Lights below for details.
Preserve Hierarchy	Always create an explicit prefab root, even if this model only has a single root. Normally, the FBX Importer strips any empty root nodes from the model as an optimization strategy. However, if you have multiple FBX files with portions of the same hierarchy you can use this option to preserve the original hierarchy. For example, file1.fbx contains a rig and a Mesh and file2.fbx contains the same rig but only the animation for that rig. If you import file2.fbx without enabling this option, Unity strips the root node, the hierarchies don't match, and the animation breaks.

Importing visibility

Unity can read visibility properties from FBX files with the **Import Visibility** property. Values and **animation curves** can enable or disable MeshRenderer components by controlling the Renderer.enabled property.

Visibility inheritance is true by default but can be overridden. For example, if the visibility on a parent **Mesh** is set to 0, all of the renderers on its children are also disabled. In this case, one animation curve is created for each child's `Renderer.enabled` property.

Some 3D modeling applications either do not support or have limitations regarding visibility properties. For more information, see:

Importing objects from Autodesk® Maya®
Importing objects from Cinema 4D
Importing objects from Blender

Importing cameras

Unity supports the following properties when importing Cameras from an .FBX file:

Property:	**Function:**
Projection mode	Orthographic or perspective. Does not support animation.
Field of View	Supports animation.
All **Physical Camera** properties	If you import a Camera with Physical Properties (for example, from from Maya), Unity creates a Camera with the **Physical Camera** property enabled and the **Focal Length**, **Sensor Type**, **Sensor Size**, **Lens Shift**, and **Gate Fit** values from the FBX file. ***Note:*** Not all 3D modeling applications have a concept of *Gate Fit*. When not supported by the 3D modeling application, the default value in Unity is **None**.
Near and **Far Clipping Plane** distance	Unity does not import any animation on these properties. When exporting from 3ds Max, enable the **Clip Manually** setting; otherwise the default values are applied on import.
Target Cameras	If you import a Target Camera, Unity creates a camera with a LookAt constraint using the target object as the source.

Importing lights

The following Light types are supported:

Omni
Spot
Directional
Area

The following Light properties are supported:

Property:	Function:
Range	The **FarAttenuationEndValue** is used if **UseFarAttenuation** is enabled. **FarAttenuationEndValue** does not support animation.
Color	Supports animation.
Intensity	Supports animation.
Spot Angle	Supports animation. Only available for Spot lights.

Note: In 3ds Max, the exported default value is the value of the property at the current selected frame. To avoid confusion, move the playhead to frame 0 when exporting.

Limitations

Some 3D modeling applications apply scaling on light properties. For instance, you can scale a spot light by its hierarchy and affect the light cone. Unity does not do this, which may cause lights to look different in Unity.

The FBX format does not define the width and height of area lights. Some 3D modeling applications don't have this property and only allow you to use scaling to define the rectangle area. Because of this, area lights always have a size of 1 when imported.

Targeted light animations are not supported unless their animation is baked.

Meshes property section

Property	*Function*
Mesh Compression	Set the level of compression ratio to reduce the file size of the Mesh. Increasing the compression ratio lowers the precision of the Mesh by using the mesh bounds and a lower bit depth per component to compress the mesh data. It's best to turn it up as high as possible without the Mesh looking too different from the uncompressed version. This is useful for optimizing game size.
Off	Don't use **compression**.
Low	Use a low compression ratio.
Medium	Use a medium compression ratio.
High	Use a high compression ratio.
Read/Write Enabled	When this option is enabled, Unity uploads the Mesh data to GPU-addressable memory, but also keeps it in CPU-addressable memory. This means that Unity can access the mesh data at run time, and you can access it from your scripts. For example, if you're generating a mesh procedurally, or if you want to copy some data from a mesh. When this option is disabled, Unity uploads the Mesh data to GPU-addressable memory, and then removes it from CPU-addressable memory. In most cases, you should disable this option to save runtime memory usage. For information on when to enable Read/Write Enabled, see Mesh.isReadable.
Optimize Mesh	Let Unity determine the order in which triangles are listed in the Mesh. Unity reorders the vertices and indices for better GPU performance.

Property	*Function*
Generate Colliders	Enable to import your Meshes with **Mesh Colliders** automatically attached. This is useful for quickly generating a **collision** Mesh for environment geometry, but should be avoided for geometry you are moving.

Geometry properties section

Property	*Function*
Keep Quads	Enable this to stop Unity from converting polygons that have four vertices to triangles. For example, if you are using Tessellation Shaders, you may want to enable this option because tessellating quads may be more efficient than tessellating polygons. Unity can import any type of polygon (triangle to N-gon). Polygons that have *more* than four vertices are always converted to triangles regardless of this setting. However, if a mesh has quads and triangles (or N-gons that get converted to triangles), Unity creates two submeshes to separate quads and triangles. Each submesh contains either triangles only or quads only. **Tip:** If you want to import quads into Unity from 3ds Max, you have to export it as an Editable Poly.
Weld Vertices	Combine vertices that share the same position in space, provided that they share the same properties overall (including, UVs, Normals, Tangents, and VertexColor). This optimizes the vertex count on Meshes by reducing their overall number. This option is enabled by default. In some cases, you might need to switch this optimization off when importing your Meshes. For example, if you intentionally have duplicate vertices which occupy the same position in your Mesh, you may prefer to use scripting to read or manipulate the individual vertex and triangle data.
Index Format	Define the size of the Mesh index buffer. **Note:** For bandwidth and memory storage size reasons, you generally want to keep **16 bit** indices as default, and only use **32 bit** when necessary, which is what the **Auto** option uses.
Auto	Let Unity decide whether to use 16 or 32 bit indices when importing a Mesh, depending on the Mesh vertex count. This is the default for Assets added in Unity 2017.3 and onwards.
16 bit	Use 16 bit indices when importing a Mesh. If the Mesh is larger, then it is split into <64k vertex chunks. This was the default setting for Projects made in Unity 2017.2 or previous.
32 bit	Use 32 bit indices when importing a Mesh. If you are using GPU-based **rendering** pipelines (for example with compute shader triangle culling), using 32 bit indices ensures that all Meshes use the same index format. This reduces **shader** complexity, because they only need to handle one format.
Normals	Defines if and how normals should be calculated. This is useful for optimizing game size.
Import	Import normals from the file. This is the default option.
Calculate	Calculate normals based on **Normals Mode**, **Smoothness Source**, and **Smoothing Angle** (below).
None	Disable normals. Use this option if the Mesh is neither normal mapped nor affected by realtime lighting.
Blend Shape Normals	Defines if and how normals should be calculated for blend shapes. Use the same values as for the **Normals** property.
Normals Mode	Define how the normals are calculated by Unity. This is only available when **Normals** is set to **Calculate**.
Unweighted Legacy	The legacy method of computing the normals (prior to version 2017.1). In some cases it gives slightly different results compared to the current implementation. It is the default for all FBX **prefabs** imported before the migration of the Project to the latest version of Unity.
Unweighted	The normals are not weighted.

Property	*Function*
Area Weighted	The normals are weighted by face area.
Angle Weighted	The normals are weighted by the vertex angle on each face.
Area and Angle Weighted	The normals are weighted by both the face area and the vertex angle on each face. This is the default option.
Smoothness Source	Set how to determine the smoothing behavior (which edges should be smooth and which should be hard).
Prefer Smoothing Groups	Use smoothing groups from the Model file, where possible.
From Smoothing Groups	Use smoothing groups from the Model file only.
From Angle	Use the **Smoothing Angle** value to determine which edges should be smooth.
None	Don't split any vertices at hard edges.
Smoothing Angle	Control whether vertices are split for hard edges: typically higher values result in fewer vertices. **Note:** Use this setting only on very smooth organics or very high poly models. Otherwise, you are better off manually smoothing inside your 3D modeling software and then importing with the **Normals** option set to **Import** (above). Since Unity bases hard edges on a single angle and nothing else, you might end up with smoothing on some parts of the Model by mistake. Only available if **Normals** is set to **Calculate**.
Tangents	Define how vertex tangents should be imported or calculated. This is only available when **Normals** is set to **Calculate** or **Import**.
Import	Import vertex tangents from FBX files if **Normals** is set to **Import**. If the Mesh has no tangents, it won't work with normal-mapped shaders.
Calculate Tangent Space	Calculate tangents using MikkTSpace. This is the default option if **Normals** is set to **Calculate**.
Calculate Legacy	Calculate tangents with legacy algorithm.
Calculate Legacy - Split Tangent	Calculate tangents with legacy algorithm, with splits across UV charts. Use this if normal map lighting is broken by seams on your Mesh. This usually only applies to characters.
None	Don't import vertex tangents. The Mesh has no tangents, so this doesn't work with normal-mapped shaders.
Swap UVs	Swap UV channels in your Meshes. Use this option if your diffuse Texture uses UVs from the **lightmap**. Unity supports up to eight UV channels but not all 3D modeling applications export more than two.
Generate Lightmap UVs	Creates a second UV channel for Lightmapping. See documentation on Lightmapping for more information.

2018–09–26 Page amended with limited editorial review

2018–08–23 Page amended with limited editorial review

2018–04–25 Page amended with limited editorial review

2017–12–05 Page amended with limited editorial review

2017–09–04 Page amended with limited editorial review

Existing (pre Unity 5.6) functionality of **Keep Quads** first documented in User Manual 5.6

Normals Mode, **Light** and ~~**Camera**~~ import options added in Unity 2017.1

~~**Materials**~~ tab added in 2017.2

Index Format property added in 2017.3

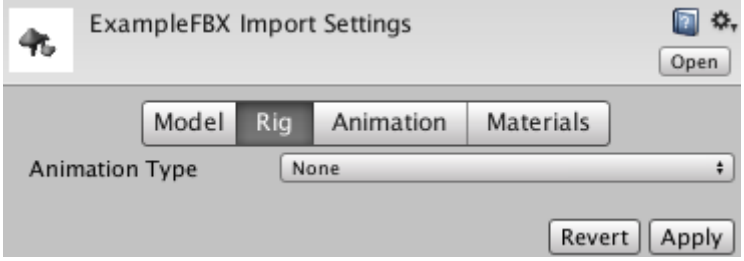
Updated description of read/write enabled setting and reorganized properties table, to match improvements in Unity 2018.3

Rig tab

Leave feedback

The settings on the **Rig** tab define how Unity maps the deformers to the **Mesh** in the imported Model so that you can animate it. For Humanoid characters, this means assigning or creating an Avatar. For non-Humanoid (*Generic*) characters, this means identifying a Root bone in the skeleton.

By default, when you select a Model in the **Project** view, Unity determines which **Animation Type** best matches the selected Model and displays it in the **Rig** tab. If Unity has never imported the file, the Animation Type is set to **None**:



No rig mapping

Property:	*Function:*
Animation Type	Specfiy the type of animation.
None	No animation present
Legacy	Use the Legacy Animation System. Import and use animations as with Unity version 3.x and earlier.
Generic	Use the Generic Animation System if your rig is *non-humanoid* (quadruped or any entity to be animated). Unity picks a root node but you can identify another bone to use as the **Root node** instead.
Humanoid	Use the Humanoid Animation System if your rig is *humanoid* (it has two legs, two arms and a head). Unity usually detects the skeleton and maps it to the Avatar correctly. In some cases, you may need to set a change the **Avatar Definition** and **Configure** the mapping manually.

Generic animation types

Your rig is *non-humanoid* (quadruped or any entity to be animated)

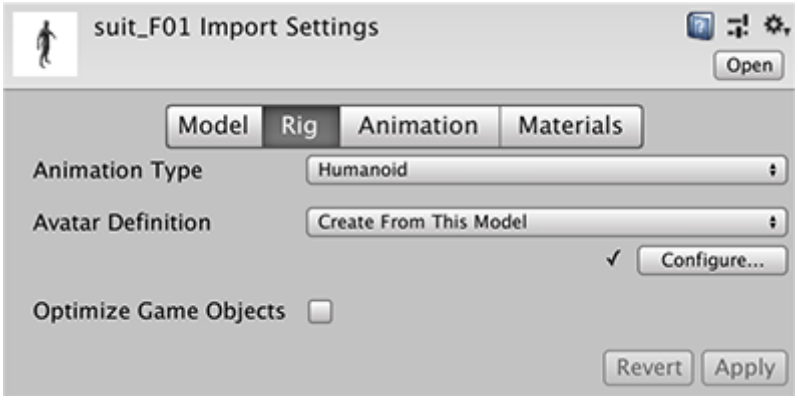
Generic Animations do not use Avatars like **Humanoid animations** do. Since the skeleton can be arbitrary, you must specify which bone is the **Root node**. The Root node allows Unity to establish consistency between

Animation clips for a generic model, and blend properly between Animations that have not been authored "in place" (that is, where the whole model moves its world position while animating).

Specifying the root node helps Unity determine between movement of the bones relative to each other, and motion of the Root node in the world (controlled from OnAnimatorMove).

Property:	*Function:*
Avatar Definition	Choose where to get the Avatar definition.
Create from this model	Create an Avatar based on this model
Copy from Other Avatar	Point to an Avatar set up on another model.
Root node	Select the bone to use as a root node for this Avatar. Only available if the **Avatar Definition** is set to **Create From This Model**.
Source	Copy another Avatar with an identical rig to import its animation clips. Only available if the **Avatar Definition** is set to **Copy from Other Avatar**.
Optimize Game Object	Remove and store the GameObject transform hierarchy of the imported character in the Avatar and Animator component. If enabled, the SkinnedMeshRenderers of the character use the Unity animation system's internal skeleton, which improves the performance of the animated characters. Only available if the **Avatar Definition** is set to **Create From This Model**. Enable this option for the final product. **Note**: In optimized mode, skinned mesh matrix extraction is also multi-threaded.

Humanoid animation types

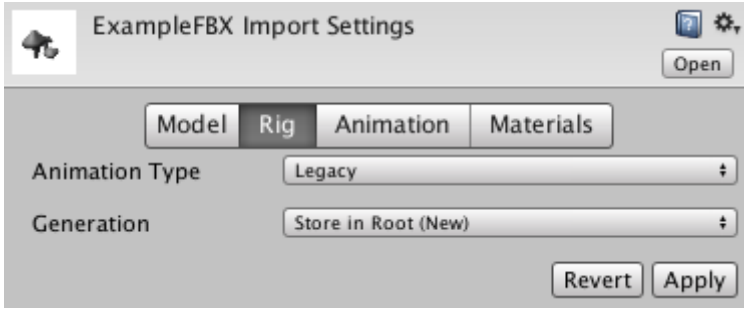


Your rig is *humanoid* (it has two legs, two arms and a head)

With rare exceptions, humanoid models have the same basic structure. This structure represents the major articulated parts of the body: the head and limbs. The first step to using Unity's Humanoid animation features is to set up and configure an **Avatar**. Unity uses the Avatar to map the simplified humanoid bone structure to the actual bones present in the Model's skeleton.

Property:	*Function:*
Avatar Definition	Choose where to get the Avatar definition.
Create from this model	Create an Avatar based on this model
Copy from Other Avatar	Point to an Avatar set up on another model.
Source	Copy another Avatar with an identical rig to import its animation clips. Only available if the **Avatar Definition** is set to **Copy from Other Avatar**.
Configure...	Open the Avatar configuration. Only available if the **Avatar Definition** is set to **Create From This Model**.
Optimize Game Object	Remove and store the GameObject transform hierarchy of the imported character in the Avatar and Animator component. If enabled, the SkinnedMeshRenderers of the character use the Unity animation system's internal skeleton, which improves the performance of the animated characters. Only available if the **Avatar Definition** is set to **Create From This Model**. Enable this option for the final product. **Note**: In optimized mode, skinned mesh matrix extraction is also multi-threaded.

Legacy animation types



Your rig uses the Legacy Animation System

Property:	*Function:*
Generation	Select the animation import method.
Don't Import	Do not import animation

Property:	*Function:*
Store in Original Roots (Deprecated)	Deprecated. Do not use.
Store in Nodes (Deprecated)	Deprecated. Do not use.
Store in Root (Deprecated)	Deprecated. Do not use.
Store in Root (New)	Import the animation and store it in the Model's root node. This is the default setting.

2018–04–25 Page amended with limited editorial review

2017–12–05 Page amended with limited editorial review

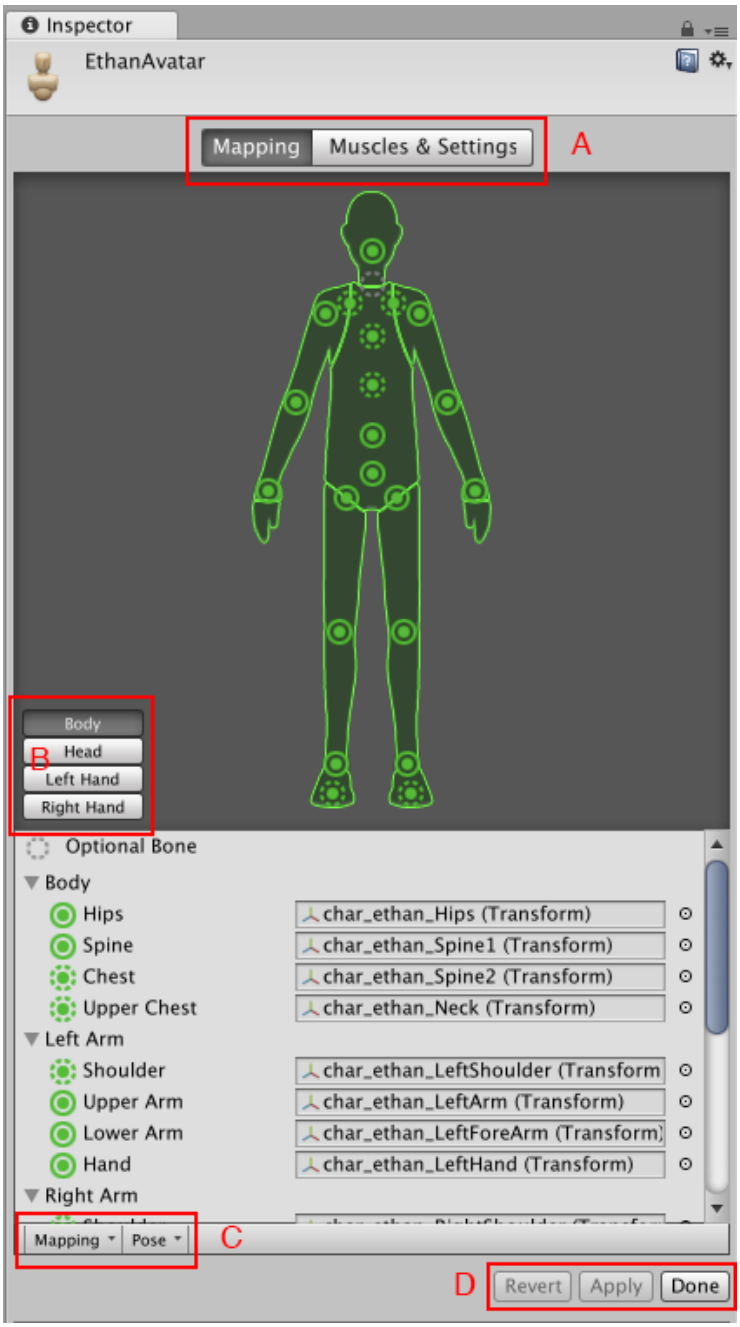
~~**Materials**~~ tab added in 2017.2

Avatar Mapping tab

Leave feedback

SWITCH TO SCRIPTING

After you save the **scene**, the **Avatar Mapping** tab appears in the **Inspector** displaying Unity's bone mapping:



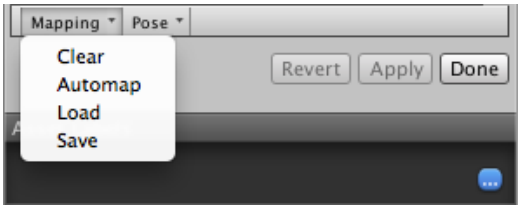
The Avatar window displays the bone mapping
(A) Buttons to toggle between the **Mapping** and **Muscles & Settings** tabs. You must **Apply** or **Revert** any changes made before switching between tabs.
(B) Buttons to switch between the sections of the Avatar: **Body**, **Head**, **Left Hand**, and **Right Hand**.
(C) Menus which provide various **Mapping** and **Pose** tools to help you map the bone structure to the Avatar.
(D) Buttons to accept any changes made (**Accept**), discard any changes (**Revert**), and leave the Avatar window (**Done**). You must **Apply** or **Revert** any changes made before leaving the **Avatar** window.

The Avatar Mapping indicates which of the bones are required (solid circles) and which are optional (dotted circles). Unity can interpolate optional bone movements automatically.

Saving and reusing Avatar data (Human Template files)

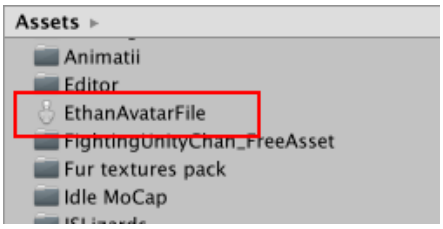
You can save the mapping of bones in your skeleton to the Avatar on disk as a Human Template file (extention `*.ht`). You can reuse this mapping for any character. For example, you want to put the Avatar mapping under source control and you prefer to commit text-based files; or perhaps you want to parse the file with your own custom tool.

To save the Avatar data in a **Human Template** file, choose **Save** from the **Mapping** drop-down menu at the bottom of the **Avatar** window.



The **Mapping** drop-down menu at the bottom of the **Avatar** window

Unity displays a dialog box for you to choose the name and location of the file to save.



To load a Human Template file previously created, choose **Mapping** > **Load** and selecr the file you want to load.

Using Avatar Masks

Sometimes it is useful to restrict an animation to specific body parts. For example, an walking animation might involve the character swaying their arms, but if they pick up a torch, they should hold it up to cast light. You can use an **Avatar Body Mask** to specify which parts of a character an animation should be restricted to. See documentation on Avatar Masks for further details.

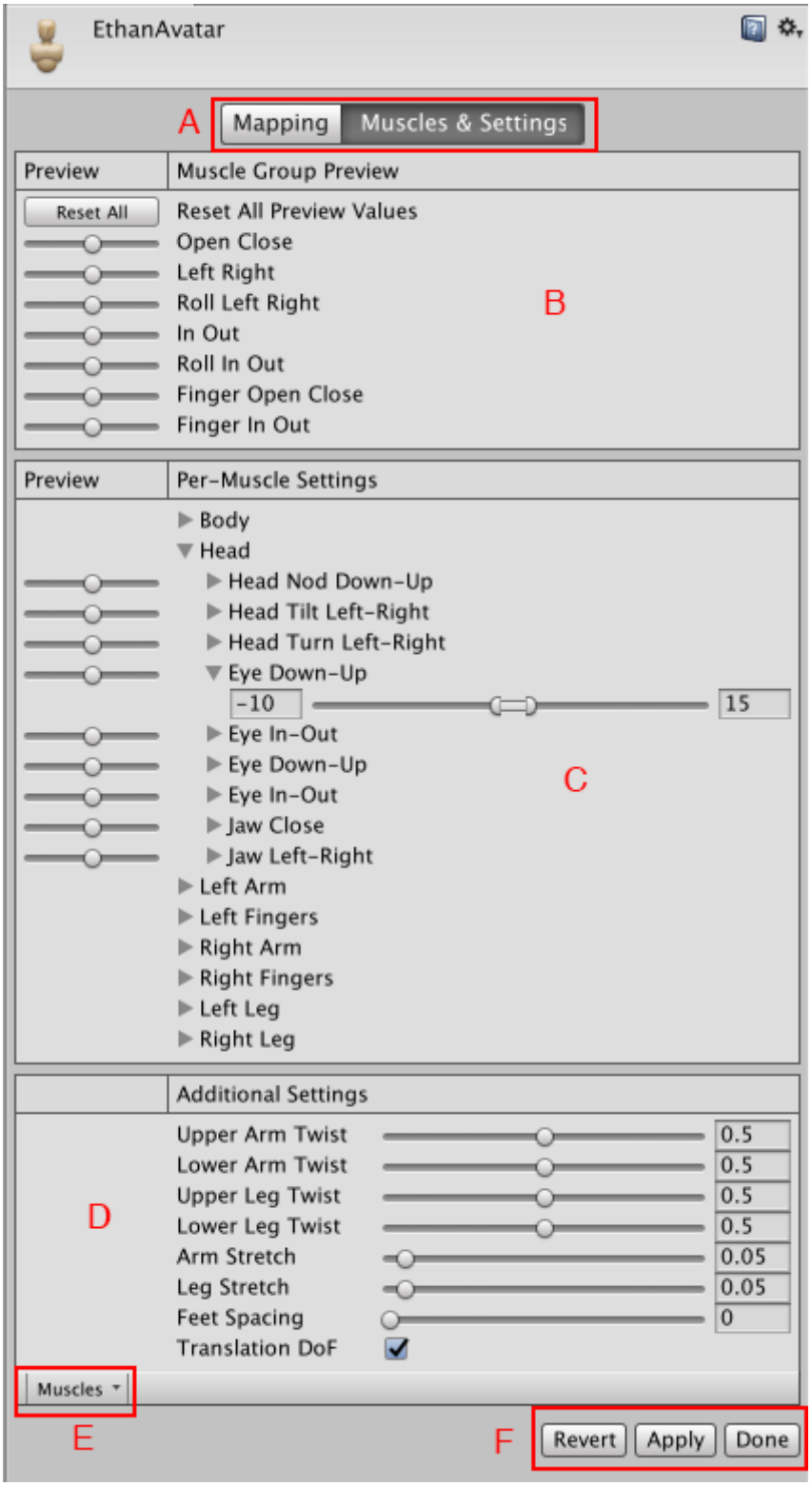
2018–04–25 Page amended with limited editorial review

Avatar Muscle & Settings tab

Leave feedback

Unity's animation system allows you to control the range of motion of different bones using **Muscles**.

Once the Avatar has been properly configured, the animation system "understands" the bone structure and allows you to start using the **Muscles & Settings** tab of the **Avatar**'s **Inspector**. Use the **Muscles & Settings** tab to tweak the character's range of motion and ensure the character deforms in a convincing way, free from visual artifacts or self-overlaps.



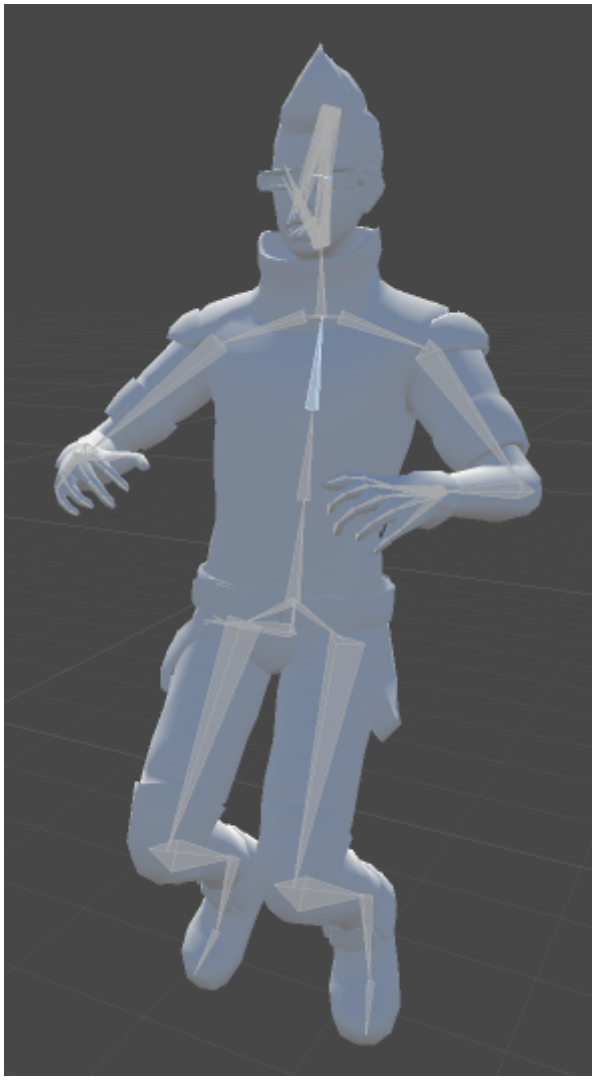
The **Muscles & Settings** tab in the **Avatar** window

The areas of the **Muscle & Settings** tab include:

(A) Buttons to toggle between the **Mapping** and **Muscles & Settings** tabs. You must **Apply** or **Revert** any changes made before switching between tabs.
(B) Use the **Muscle Group Preview** area to manipulate the character using predefined deformations. These affect several bones at once.
(C) Use the **Per-Muscle Settings** area to adjust individual bones in the body. You can expand the muscle settings to change the range limits of each settings. For example, by default, Unity gives the Head-Nod and Head-Tilt settings a possible range of –40 to 40 degrees but you can decrease these ranges even further to add stiffness to these movements.
(D) Use the **Additional Settings** to adjust specific effects in the body.
(E) The **Muscles** menu provides a **Reset** tool to return all muscle settings to their default values.
(F) Buttons to accept any changes made (**Accept**), discard any changes (**Revert**), and leave the Avatar window (**Done**). You must **Apply** or **Revert** any changes made before leaving the **Avatar** window.

Previewing changes

For the settings in the **Muscle Group Preview** and **Per-Muscle Settings** areas, you can preview the changes right in the **Scene** view. You can drag the sliders left and right to see the range of movement for each setting applied to your character:



Preview the changes to the muscles settings in the Scene view
You can see the bones of the skeleton through the **Mesh**.

Translate Degree of Freedom (DoF)

You can enable the **Translate DoF** option in the **Additional Settings** to enable translation animations for the humanoid. If this option is disabled, Unity animates the bones using only rotations. **Translation DoF** is available for Chest, UpperChest, Neck, LeftUpperLeg, RightUpperLeg, LeftShoulder and RightShoulder muscles.

Note: Enabling **Translate DoF** can increase performance requirements, because the animation system needs to perform an extra step to retarget **humanoid animation**. For this reason, you should only enable this option if you know your animation contains animated translations of some of the bones in your character.

2018–04–25 Page amended with limited editorial review

Avatar Mask window

Leave feedback

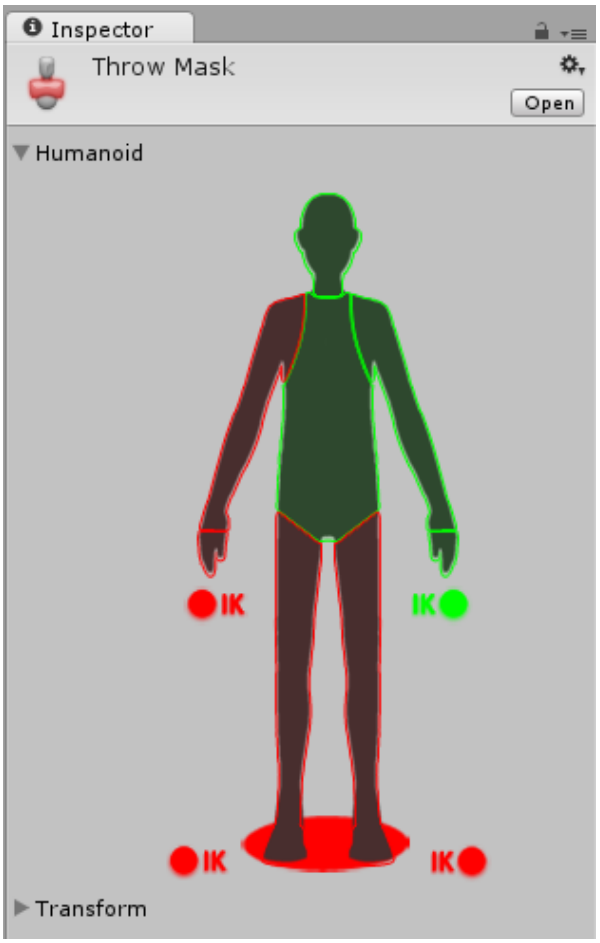
SWITCH TO SCRIPTING

There are two ways to define which parts of your animation should be masked:

By selecting from a Humanoid body map
By choosing which bones to include or exclude from a Transform hierarchy

Humanoid body selection

If your animation uses a Humanoid **Avatar**, you can select or deselect portions of the simplified humanoid body diagram to indicate where to mask the animation:



Defining an Avatar Mask using the Humanoid body
The body diagram groups body parts into these portions:

Head
Left Arm
Right Arm
Left Hand
Right Hand
Left Leg
Right Leg
Root (denoted by the “shadow” under the feet)

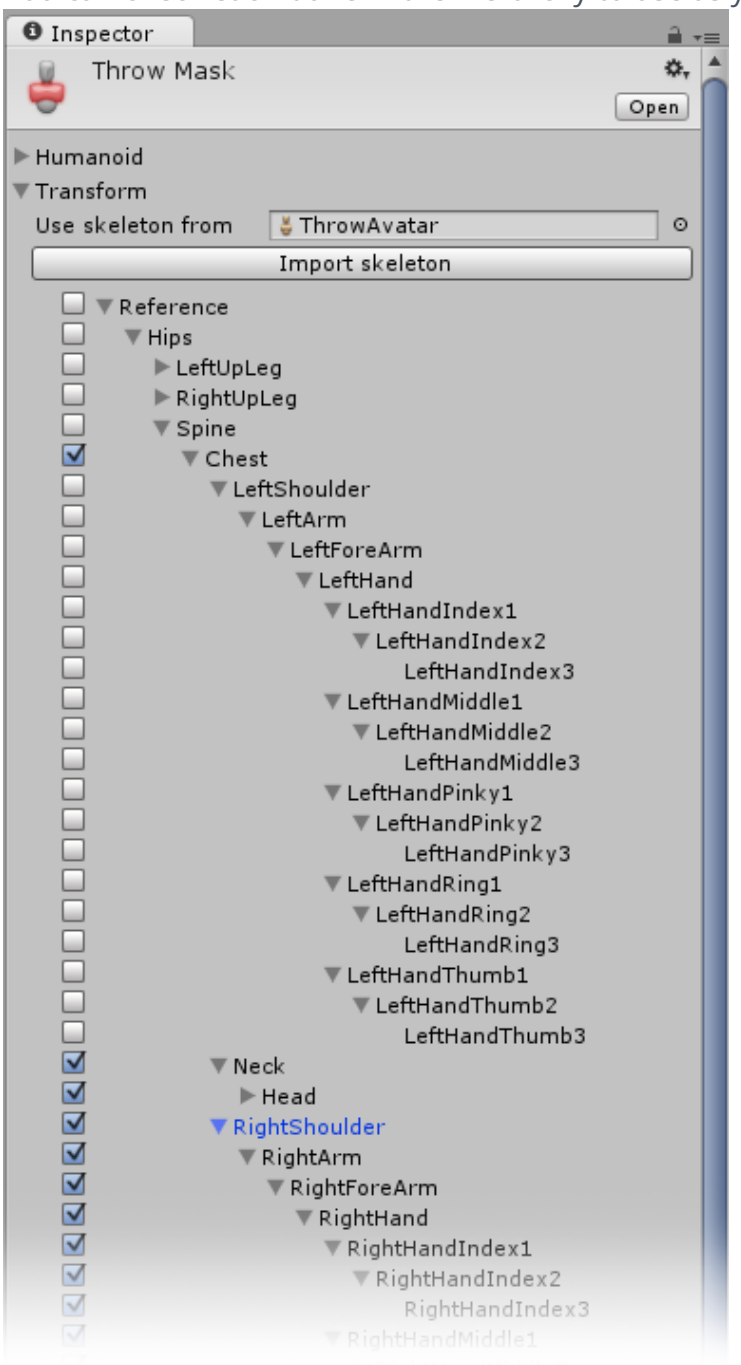
To include animation from one of these body portions, click the Avatar diagram for that portion until it appears as green. To exclude animation, click the body portion until it appears red. To include or exclude all, double-click the empty space surrounding the Avatar.

You can also toggle **Inverse** Kinematics__ (IK)__ for hands and feet, which determines whether or not to include IK curves in animation blending.

Transform selection

Alternatively, if your animation does not use a Humanoid Avatar, or if you want more detailed control over which individual bones are masked, you can select or deselect portions of the Model's hierarchy:

Assign a reference to the Avatar whose transform you would like to mask.
Click the **Import Skeleton** button. The hierarchy of the avatar appears in the **inspector**.
You can check each bone in the hierarchy to use as your mask.



Defining an avatar mask using the Transform method

Mask **assets** can be used in Animator Controllers, when specifying Animation Layers to apply masking at runtime, or in the import settings of of your animation files to apply masking during to the import animation.

A benefit of using Masks is that they tend to reduce memory overheads since body parts that are not active do not need their associated **animation curves**. Also, the unused curves need not be calculated during playback which will tend to reduce the CPU overhead of the animation.

2018–04–25 Page amended with limited editorial review

Human Template window

Leave feedback

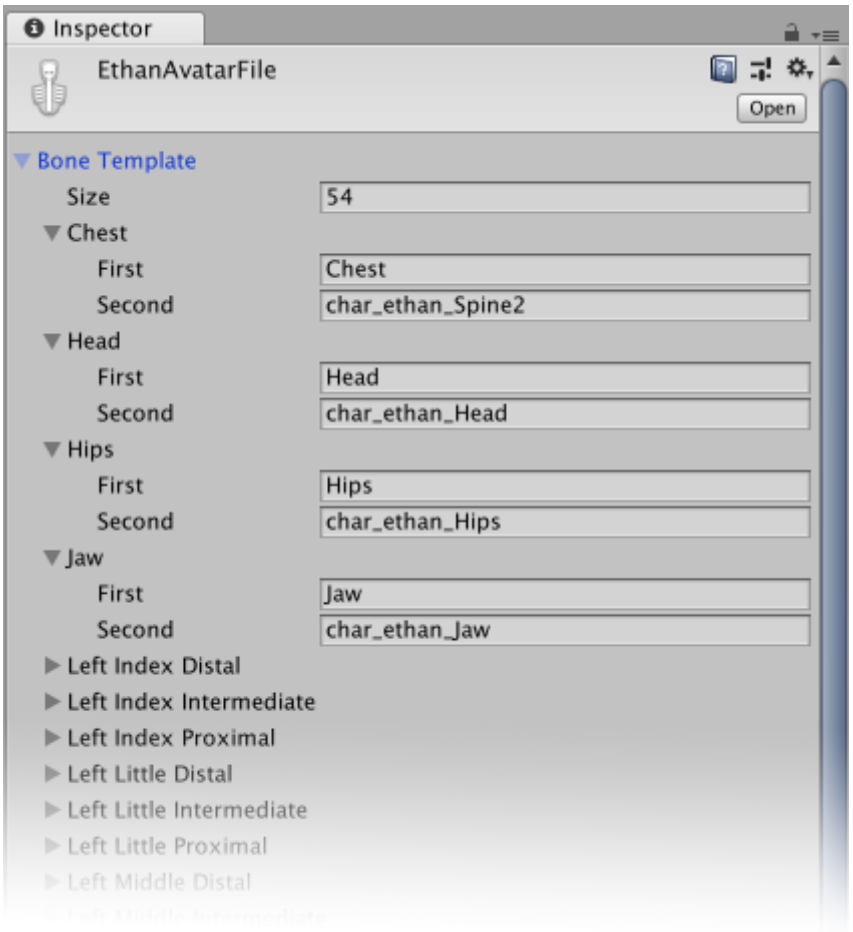
A **Human Template** file (`*.ht`) stores a Humanoid bone mapping for a Model that you saved on the Avatar window in a YAML format:

EthanAvatarFile.ht

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!1105 &110500000
HumanTemplate:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 0}
  m_Name: EthanAvatarFile
  m_BoneTemplate:
    Chest: char_ethan_Spine2
    Head: char_ethan_Head
    Hips: char_ethan_Hips
    Jaw: char_ethan_Jaw
    Left Index Distal: char_ethan_LeftHandIndex3
    Left Index Intermediate: char_ethan_LeftHandIndex2
    Left Index Proximal: char_ethan_LeftHandIndex1
    Left Little Distal: char_ethan_LeftHandPinky3
    Left Little Intermediate: char_ethan_LeftHandPinky2
    Left Little Proximal: char_ethan_LeftHandPinky1
    Left Middle Distal: char_ethan_LeftHandMiddle3
    Left Middle Intermediate: char_ethan_LeftHandMiddle2
    Left Middle Proximal: char_ethan_LeftHandMiddle1
```

A Human Template file in YAML format

The **Human Template** window displays the contents of Human Template files as standard Unity text boxes.



The Human Template window

Each grouping represents an entry in the YAML file, with the name of the bone mapping target labelled **First** and the name of the bone in the **Model file** labelled **Second**.

You can edit most of the values in this file using this property, but Unity instantly updates every change you make to the file. However, you can Undo any changes while this window is active.

2018–04–25 Page amended with limited editorial review

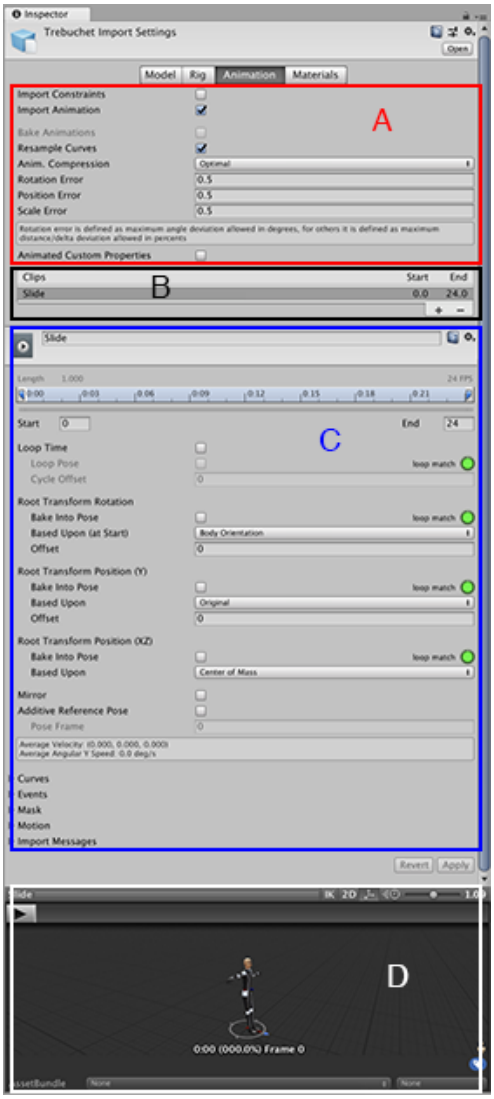
Animation tab

Leave feedback

SWITCH TO SCRIPTING

Animation Clips are the smallest building blocks of animation in Unity. They represent an isolated piece of motion, such as RunLeft, Jump, or Crawl, and can be manipulated and combined in various ways to produce lively end results (see Animation State Machines, Animator Controller, or Blend Trees). You can select Animation Clips from imported FBX data.

When you click on the model containing animation clips, these properties appear:

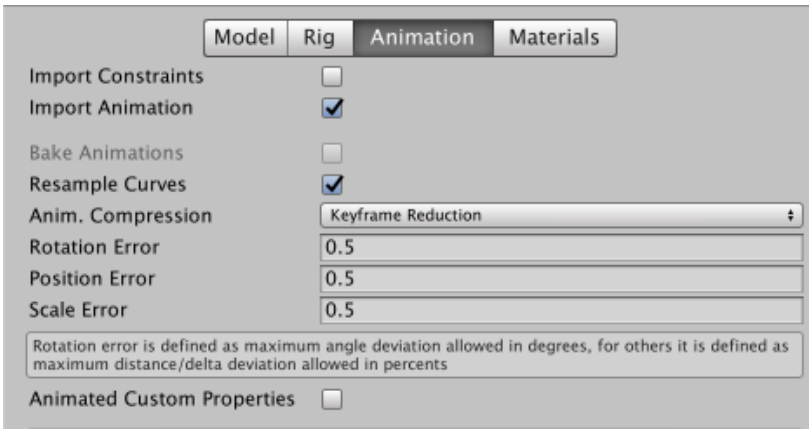


The Animation Clip Inspector

There are four areas on the **Rig** tab of the **Inspector** window:

(A) Asset-specific properties. These settings define import options for the entire **Asset**.
(B) Clip selection list. You can select any item from this list to display its properties and preview its animation. You can also define new clips.
(C) Clip-specific properties. These settings define import options for the selected **Animation Clip**.
(D) Animation preview. You can playback the animation and select specific frames here.

Asset-specific properties



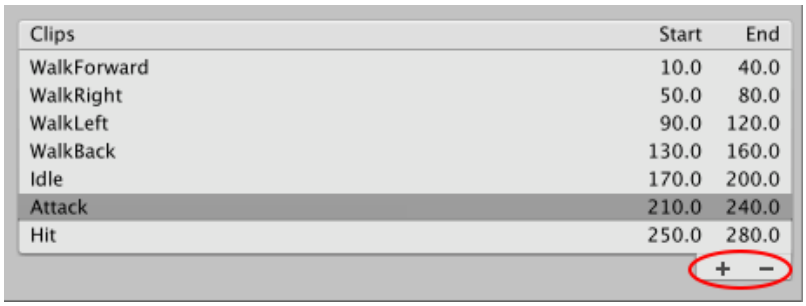
Import options for the entire Asset

These properties apply to all animation clips and constraints defined within this Asset:

Property:	*Function:*
Import Constraints	Import constraints from this asset.
Import Animation	Import animation from this asset. *Note:* If disabled, all other options on this page are hidden and no animation is imported.
Bake Animations	Bake animations created using IK or Simulation to forward kinematic keyframes. Only available for Autodesk® Maya®, Autodesk® 3ds Max® and Cinema 4D files.
Resample Curves	Resample animation curves as Quaternion values and generate a new Quaternion keyframe for every frame in the animation. This option is enabled by default. Disable this to keep animation curves as they were originally authored this only if you're having issues with the interpolation between keys in your original animation Only appears if the import file contains Euler curves.
Anim. Compression	The type of **compression** to use when importing the animation.
Off	Disable **animation compression**. This means that Unity doesn't reduce **keyframe** count on import, which leads to the highest precision animations, but slower performance and bigger file and runtime memory size. It is generally not advisable to use this option - if you need higher precision animation, you should enable **keyframe** reduction and lower allowed **Animation Compression Error** values instead.
Keyframe Reduction	Reduce redundant keyframes on import. If selected, the **Animation Compression Errors** options are displayed. This affects both file size (runtime memory) and how curves are evaluated.
Keyframe Reduction and Compression	Reduce keyframes on import and compress keyframes when storing animations in files. This affects only file size - the runtime memory size is the same as **Keyframe Reduction**. If selected, the **Animation Compression Errors** options are displayed.
Optimal	Let Unity decide how to compress, either by keyframe reduction or by using dense format. Only for **Generic** and **Humanoid** Animation Type rigs.
Animation Compression Errors	Only available when **Keyframe Reduction** or **Optimal** compression is enabled.
Rotation Error	How much to reduce rotation curves. The smaller the value, the higher the precision.

Property:	*Function:*
Position Error	How much to reduce position curves. The smaller the value, the higher the precision.
Scale Error	How much to reduce scale curves. The smaller the value, the higher the precision.
Animated Custom Properties	Import any FBX properties that you designated as custom user properties. Unity only supports a small subset of properties when importing FBX files (such as translation, rotation, scale and visibility). However, you can treat standard FBX properties like user properties by naming them in your importer script via the extraUserProperties member. During import, Unity then passes any named properties to the Asset postprocessor just like 'real' user properties.

Clip selection list

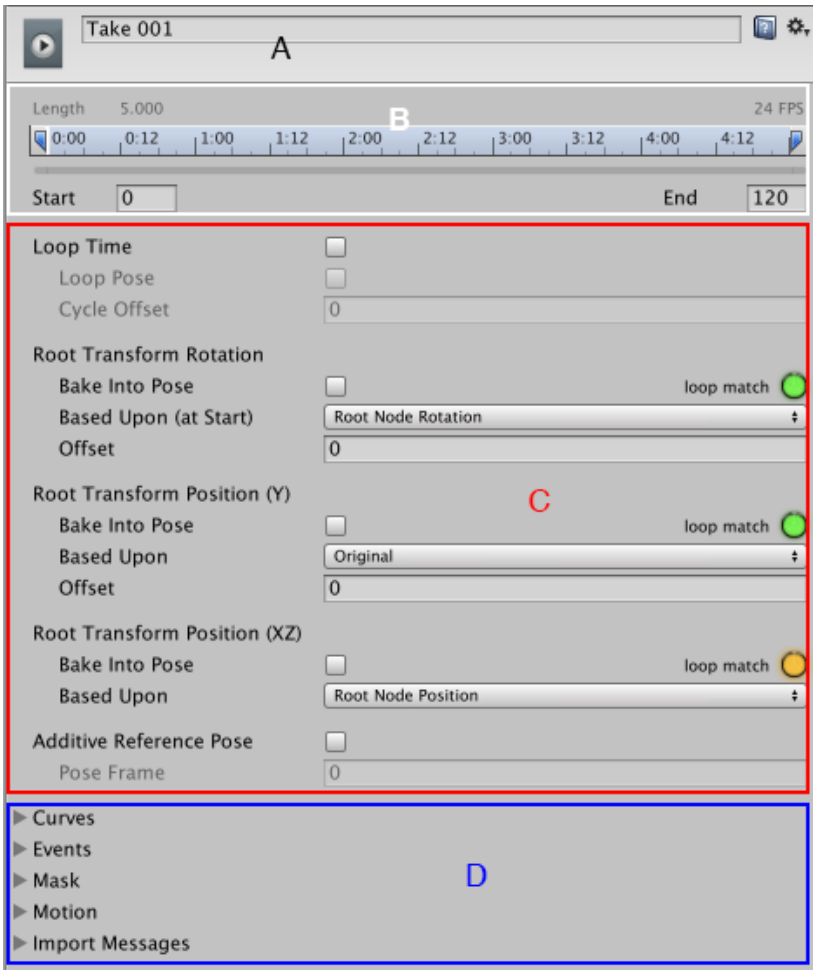


List of clips defined in this file

You can perform these tasks in this area of the **Rig** tab:

Select a clip from the list to display its clip-specific properties.
Play a selected clip in the clip preview pane.
Create a new clip for this file with the add (+) button.
Remove the selected clip definition with the delete (–) button.

Clip-specific properties



Import options for the selected Animation clip

This area of the **Rig** tab displays these features:

(A) The (editable) name of the selected clip
(B) The animation clip timeline
(C) Clip properties to control looping and pose
(D) Expandable sections for: defining curves, events, masks, and motion roots; and viewing messages from the import process

You can set these properties separately for each animation clip defined within this asset:

Property:	*Function:*
Area A (editable name)	
	The take in the source file to use as a source for this animation clip.
	This is what defines a set of animation as separated in Motionbuilder, Autodesk® Maya® and other 3D packages. Unity can import these takes as individual clips. You can create them from the whole file or from a subset of frames.
Area B (timeline features)	
	You can use the drag the start and end indicators around the **timeline** to define frame ranges for each clip.
Start	Start frame of the clip.
End	End frame of the clip.
Area C (looping and pose control)	
Loop Time	Play the animation clip through and restart when the end is reached.
Loop Pose	Loop the motion seamlessly.

Property:	*Function:*
Cycle Offset	Offset to the cycle of a looping animation, if it starts at a different time.
Root Transform Rotation	
Bake into Pose	Bake root rotation into the movement of the bones. Disable to store as **root motion**.
Based Upon	Basis of root rotation.
Original	Keep the original rotation from the source file.
Root Node Rotation	Keeps the upper body pointing forward. Only available for the **Generic** Animation Type.
Body Orientation	Keep the upper body pointing forward. Only available for the **Humanoid** Animation Type.
Offset	Offset to the root rotation (in degrees).
Root Transform Position (Y)	
Bake into Pose	Bake vertical root motion into the movement of the bones. Disable to store as root motion.
Based Upon (at Start)	Basis of vertical root position.
Original	Keep the vertical position from the source file.
Root Node Position	Use the vertical root position. Only available for the **Generic** Animation Type.
Center of Mass	Keep center of mass aligned with the root transform position. Only available for the **Humanoid** Animation Type.
Feet	Keep feet aligned with the root transform position. Only available for the **Humanoid** Animation Type.
Offset	Offset to the vertical root position.
Root Transform Position (XZ)	
Bake into Pose	Bake horizontal root motion into the movement of the bones. Disable to store as root motion.
Based Upon	Basis of horizontal root position.
Original	Keep the horizontal position from the source file.
Root Node Position	Use the horizontal root transform position. Only available for the **Generic** Animation Type.
Center of Mass	Keep aligned with the root transform position. Only available for the **Humanoid** Animation Type.
Offset	Offset to the horizontal root position.
Mirror	Mirror left and right in this clip. Only appears if the Animation Type is set to **Humanoid**.
Additive Reference Pose	Enable to set frame for the reference pose used as the base for the additive animation layer. A blue marker becomes visible in the timeline editor:

Property:	*Function:*
Pose Frame	Enter a frame number to use as the reference pose. You can also drag the blue marker in the timeline to update this value. Only available if **Additive Reference Pose** is enabled.

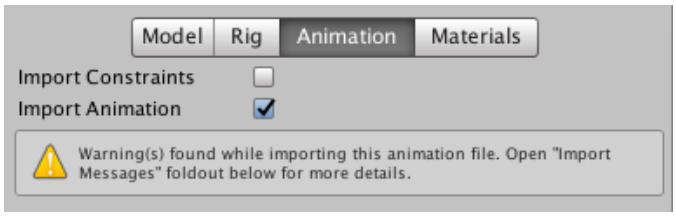
Area D (expandable sections)

Curves	Expand this section to manage animation curves on imported clips.
Events	Expand this section to manage animation events on imported clips.
Mask	Expand this section to manage masking imported clips.
Motion	Expand this section to manage selecting a root motion node.
Import Messages	Expand this section to see detailed information about how your animation was imported, including an optional *Retargeting Quality Report*.

Creating clips is essentially defining the start and end points for segments of animation. In order for these clips to loop, they should be trimmed in such a way to match the first and last frame as best as possible for the desired loop.

Animation import warnings

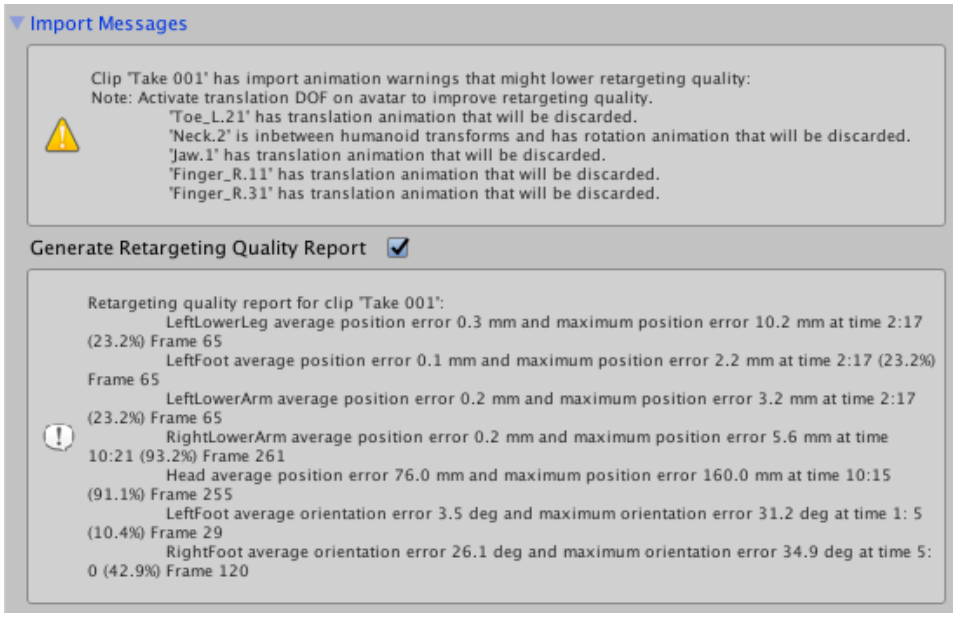
If any problems occur during the animation import process, a warning appears at the top of the Animations Import inspector:



Animation import warning messages

The warnings do not necessarily mean your animation has not imported or doesn't work. It may just mean that the imported animation looks slightly different from the source animation.

To see more information, expand the **Import Messages** section:



Animation import warning messages

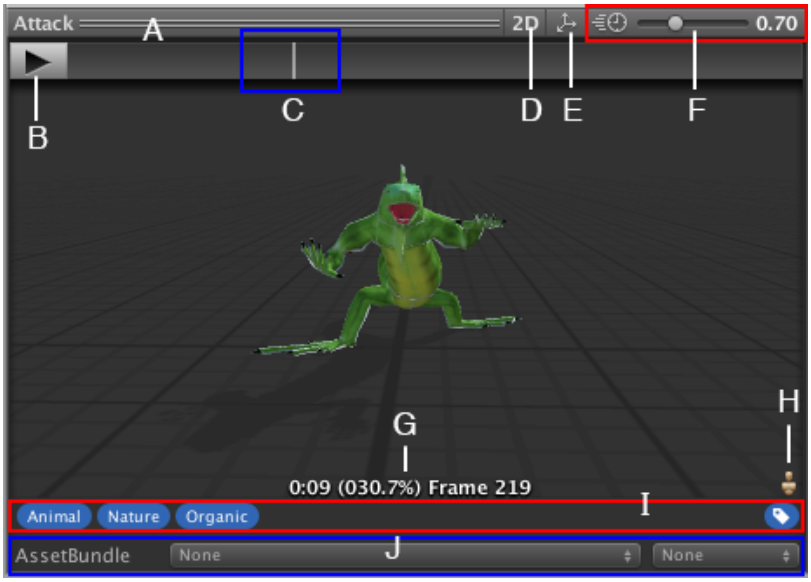
In this case, Unity has provided a **Generate Retargeting Quality Report** option which you can enable to see more specific information on the retargeting problems.

Other warning details you may see include the following:

Default bone length found in this file is different from the one found in the source **avatar**.
Inbetween bone default rotation found in this file is different from the one found in the source avatar.
Source avatar hierarchy doesn't match one found in this model.
This animation has Has translation animation that will be discarded.
Humanoid animation has inbetween transforms and rotation that will be discarded.
Has scale animation that will be discarded.
These messages indicate that some data present in your original file was omitted when Unity imported and converted your animation to its own internal format. These warnings essentially tell you that the retargeted animation may not exactly match the source animation.

Note: Unity does not support pre- and post-extrapolate modes (also known as pre- and post-infinity modes) other than **constant**, and converts these to **constant** when imported.

Animation preview



Animation clip preview

The preview area of the **Rig** tab provides these features:

(A) The name of the selected clip
(B) The play/pause button
(C) The playback head on the preview timeline (allows scrubbing back and forth)
(D) The 2D preview mode button (switches between orthographic and perspective camera)
(E) The pivot and mass center display button (switches between displaying and hiding the gizmos)
(F) The animation preview speed slider (move left to slow down; right to speed up)
(G) The playback status indicator (displays the location of the playback in seconds, percentage, and frame number)
(H) The Avatar selector (change which GameObject will preview the action)
(I) The **Tag** bar, where you can define and apply Tags to your clip
(J) The AssetBundles bar, where you can define AssetBundles and Variants

---

2018–04–25 Page amended with limited editorial review

2017–12–05 Page amended with no editorial review

Materials tab added in 2017.2

Euler curve resampling

Leave feedback

Rotations in 3D applications are usually represented as **Quaternions** or Euler angles. For the most part, Unity represents rotations as **Quaternions** internally; however, it's important to have a basic understanding of rotation and orientation in Unity.

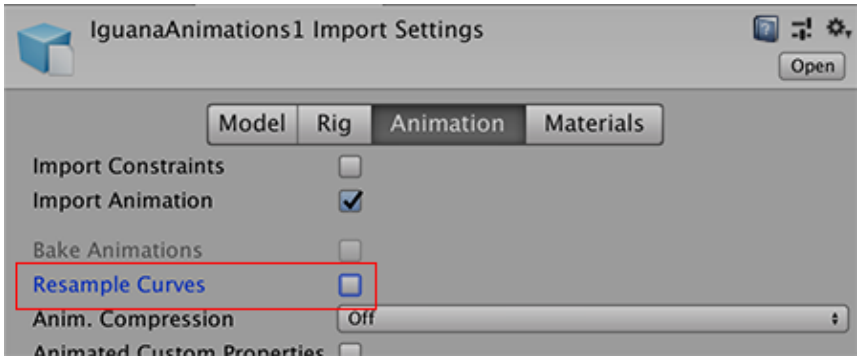
When you import files containing animation that come from external sources, the imported files usually contain **keyframe** animation in Euler format. Unity's default behavior is to resample these animations as Quaternion values and generate a new Quaternion **keyframe** for every frame in the animation. This minimizes the differences between the source animation and how it appears in Unity.

There are still some situations where the quaternion representation of the imported animation may not match the original closely enough, even with resampling. For this reason, Unity provides the option to turn off animation resampling. This means that you can use the original Euler animation keyframes at run-time instead.

Note: You should only keep the Euler curves as a last resort, if the default Quaternion **interpolation** between frames gives bad results and causes issues.

Keeping the original Euler curves on imported animations

To use the original Euler curve values in an animation file, uncheck the **Resample Curves** option in the Animation tab:



The Resample Curves option in the Animations tab

When you disable this option, Unity keeps the rotation curve with its original keyframes, in Euler or Quaternion mode as appropriate for the curve type.

Note: The FBX SDK automatically resamples any rotation curve on a **joint** that has pre- or post-rotations. This means that Unity automatically imports them as Quaternion curves.

Unity supports such a wide variety of imported files and attempts to keep the imported curves as close to the original as possible. In order to achieve this, Unity supports all normal (non-repeating) Euler rotation orders, and imports curves in their original rotation orders.

Euler values and the Unity engine

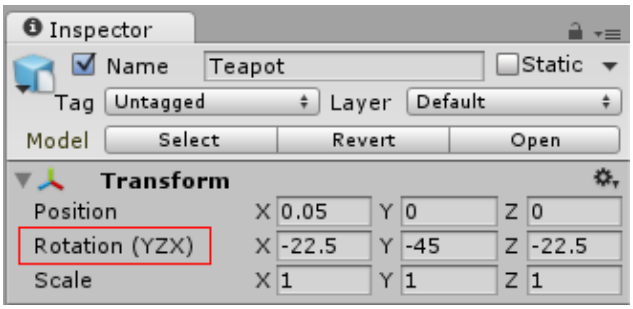
When using original Euler (non-resampled) rotations, you can see very little visual difference in the playback of animations. Under the hood, Unity stores these curves in Euler representation even at run-time. However, Unity has to convert rotation values to Quaternions eventually, since the engine only works with Quaternions.

When you disable the **Resample Curves** option, Unity keeps the rotation values as Euler values up until they are applied to a **GameObject**. This means that the end result should look as good as the original, but with an improvement in memory, since rotation curves that have not been baked in the authoring software take up less memory.

Non-default Euler orders in the Transform Inspector

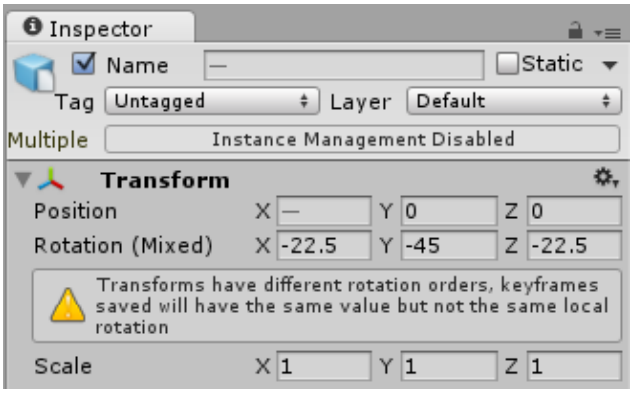
By default, Unity applies the Euler angles that appear in the Transform **Inspector** in the Z,X,Y order.

When playing back or editing imported animations that feature Euler curves with a rotation order different from Unity's default, Unity displays an indicator of the difference next to the rotation fields:



The inspector showing that a non-default rotation order is being used for the rotation animation on this object

When editing multiple transforms with differing rotation orders, Unity displays a warning message that the same Euler rotation applied will give different results on curves with different rotation orders:



The inspector showing that a mixture of rotation orders are being used for the rotation animation on the selected group of objects

---

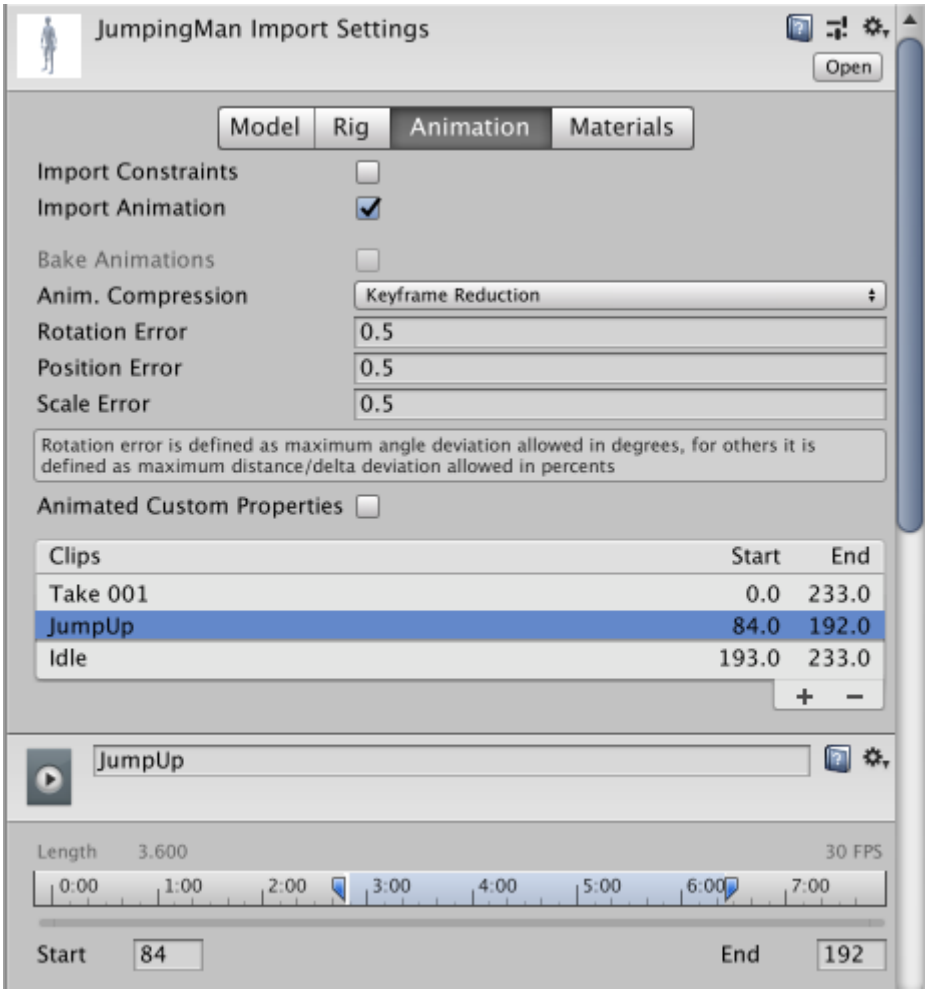
2018–04–25 Page amended with limited editorial review

Extracting animation clips

Leave feedback

An animated character typically has a number of different movements that are activated in the game in different circumstances, called Animation Clips. For example, we might have separate animation clips for walking, running, jumping, throwing, and dying. Depending on how the artist set up the animation in the 3D modeling application, these separate movements might be imported as distinct animation clips or as one single clip where each movement simply follows on from the previous one. In cases where there is only one long clip, you can extract **component** animation clips inside Unity, which adds a few extra steps to your workflow.

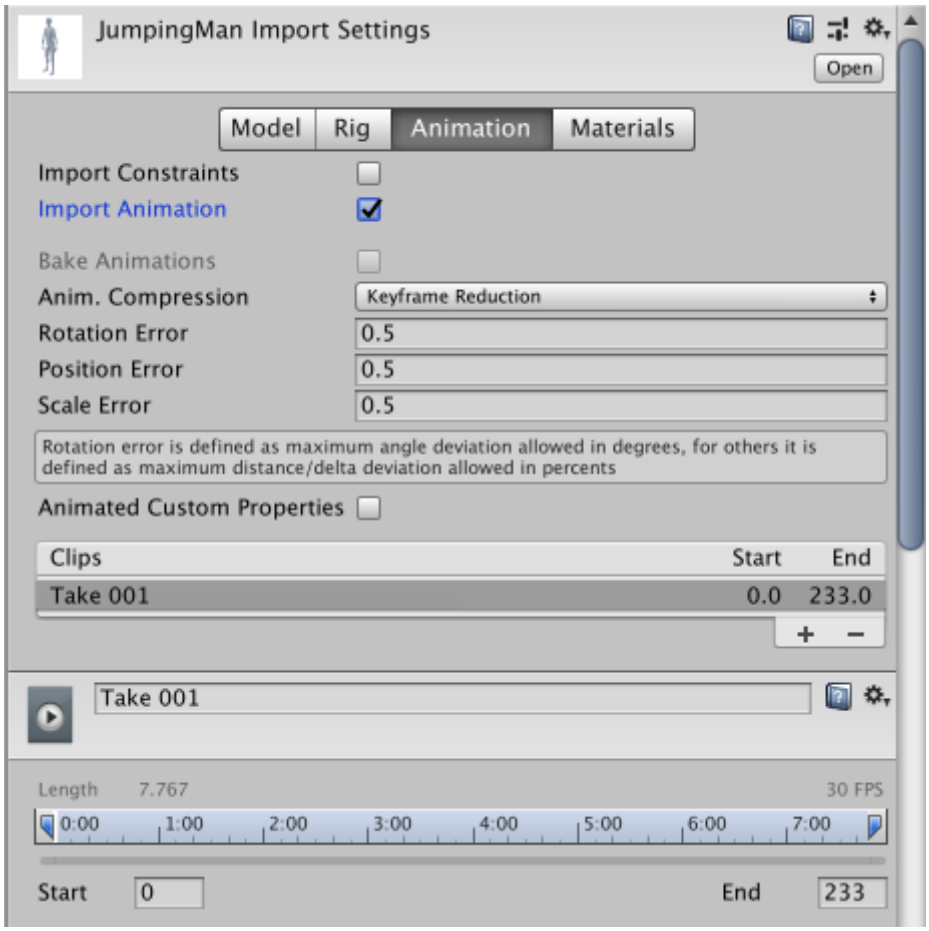
If your model has multiple animations that you already defined as individual clips, the Animations tab looks like this:



Model file with several animation clips defined

You can preview any of the clips that appear in the list. If you need to, you can edit the time ranges of the clips.

If your model has multiple animations supplied as one continuous take, the **Animation** tab looks like this:



Model file with one long continous animation clip

In this case, you can define the time ranges (frames or seconds) that correspond to each of the separate animation sequences (walking, jumping, running, and idling). You can create a new animation clip by following these steps:

Click the add (+) button.
Select the range of frames or seconds that it includes.
You can also change the name of the clip.

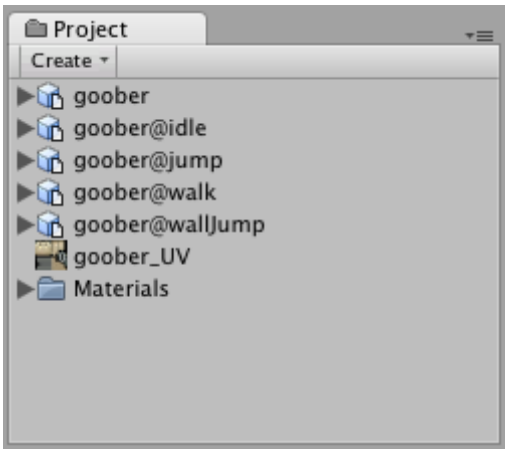
For example, you could define the following:

idle animation during frames 0 - 83
jump animation during frames 84 - 192
slightly swinging arms animation during frames 193 - 233

For further information, see the Animation tab.

Importing animations using multiple model files

Another way to import animations is to follow a naming scheme that Unity allows for the animation files. You can create separate **model files** and name them with the convention `modelName@animationName.fbx`. For example, for a model called `goober`, you could import separate idle, walk, jump and walljump animations using files named `goober@idle.fbx`, `goober@walk.fbx`, `goober@jump.fbx` and `goober@walljump.fbx`. When exporting animation like this, it is unnecessary to include the **Mesh** in these files, but in that case you should enable the Preserve Hierarchy Model import option.



An example of four animation files for an animated character

Unity automatically imports all four files and collects all animations to the file without the @ sign in. In the example above, Unity imports the `goober.mb` file with references to the `idle`, `jump`, `walk` and `wallJump` animations automatically.

For FBX files, you can export the Mesh in a Model file without its animation. Then export the four clips as `goober@_animname_.fbx` by exporting the desired **keyframes** for each (enable animation in the FBX dialog).

2018–04–25 Page amended with limited editorial review

Loop optimization on Animation clips

Leave feedback

A common operation for people working with animations is to make sure they loop properly. For example, if a character is walking down a path, the walking motion comes from an **Animation clip**. The motion might last for only 10 frames but that motion plays in a continuous loop. In order to make the walking motion seamless, it must begin and end in a similar pose. This ensures there is no foot sliding or strange jerky motions.

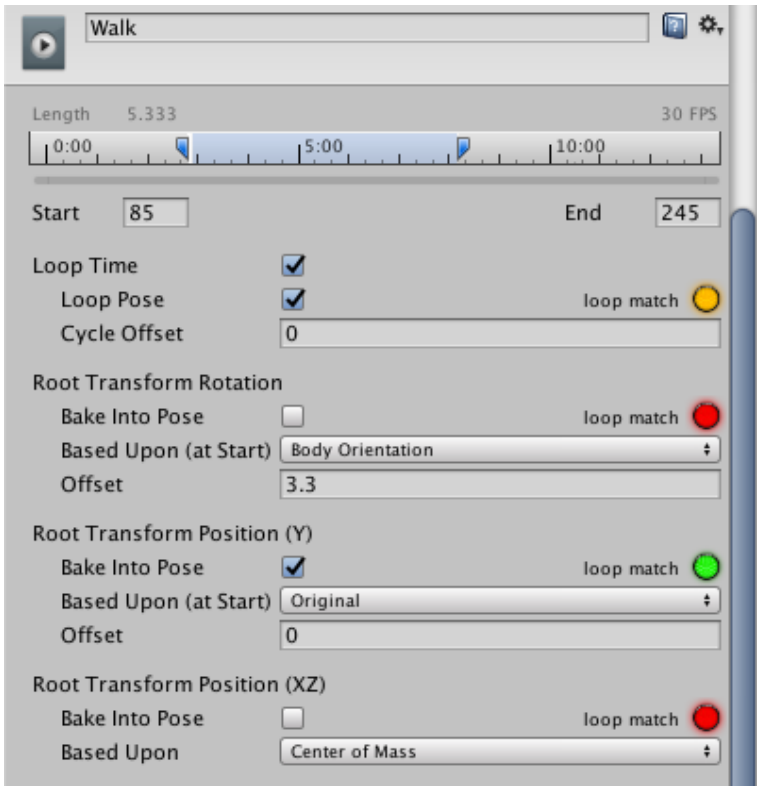
Animation clips can loop on pose, rotation, and position. Using the example of the walk cycle, you want the start and end points for **Root Transform** Rotation and **Root Transform** Position in Y to match. You don't want to match the start and end points for the **Root Transform** Position in XZ, because your character would never get anywhere if its feet keep returning to their horizontal pose.

Unity provides match indicators and a set of special loop optimization graphs under the clip-specific import settings on the Animation tab. These provide visual cues to help you optimize where to clip the motion for each value.

To optimize whether the looping motion begins and ends optimally, you can view and edit the looping match curves.

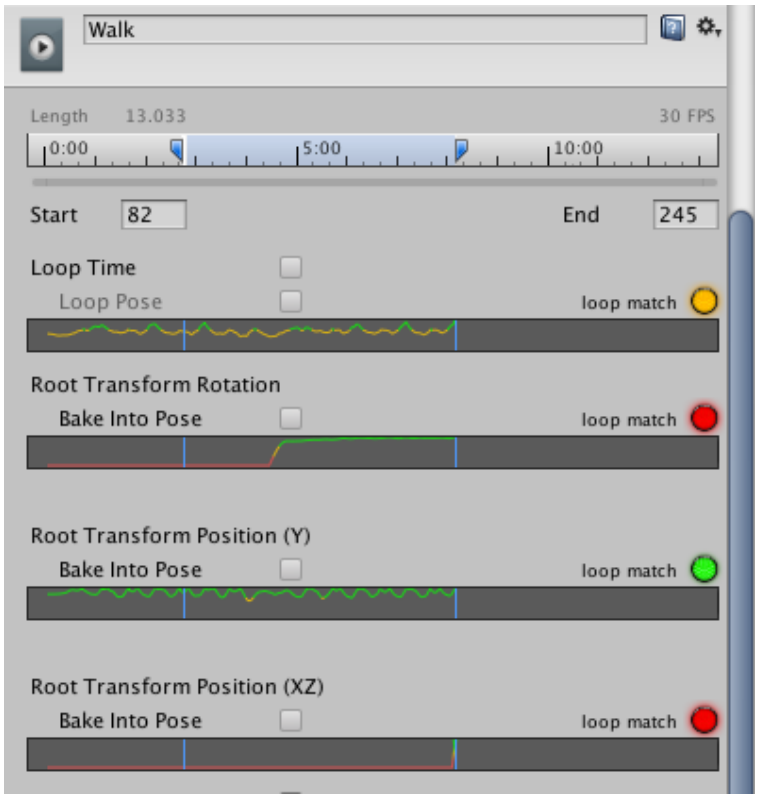
Viewing loop optimization graphs

In this example, the looping motion displays bad matches for the clip ranges, shown by the red and yellow indicators:



Red and yellow indicators show bad matches for looping

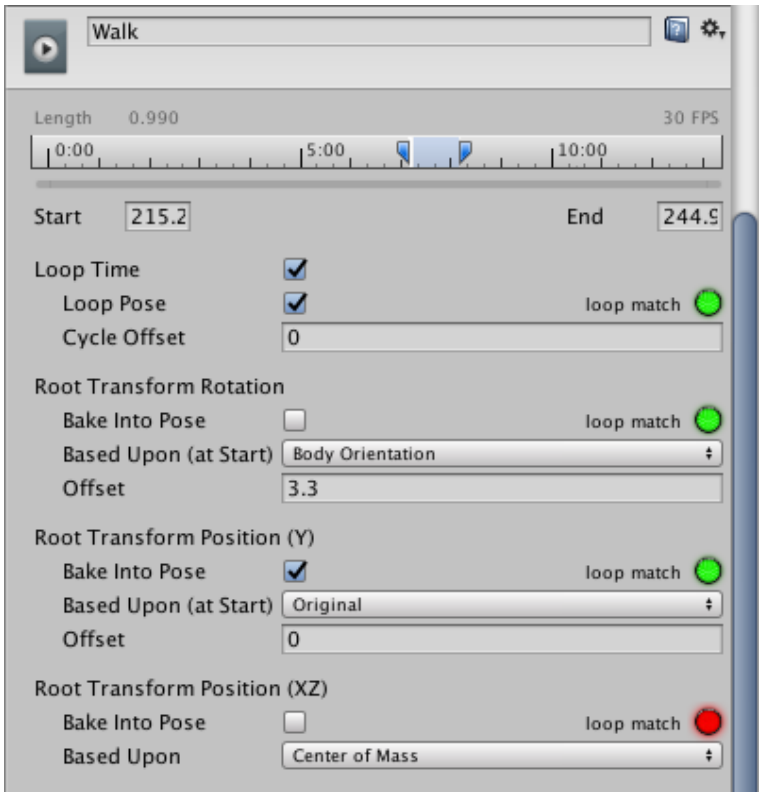
To see the loop optimization graphs, click and hold either the start or end indicator on the **timeline**. The **Based Upon** and **Offset** values disappear and one curve for each loop basis appears:



Looping graphs for bad matches

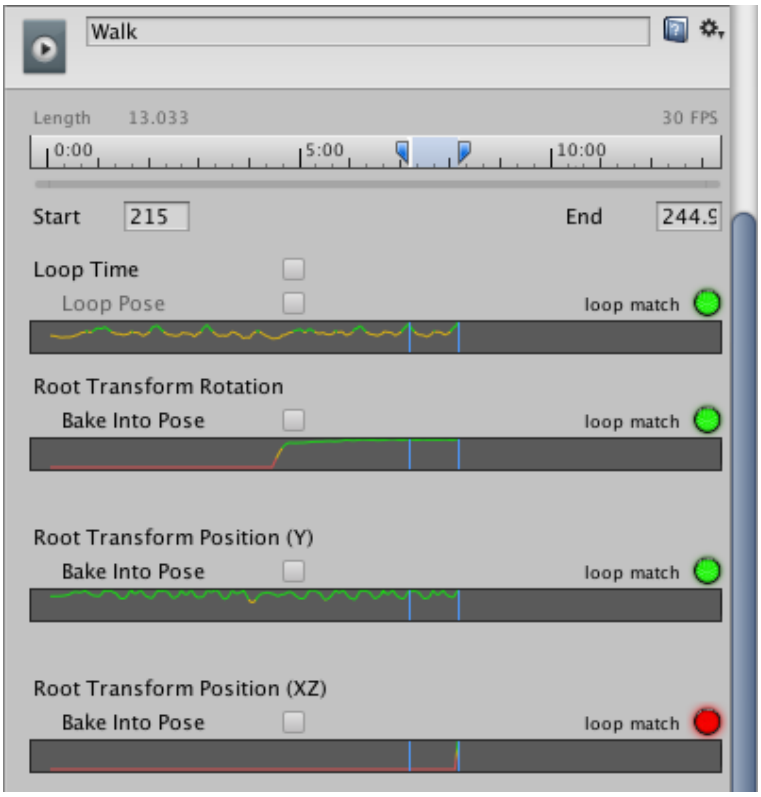
Optimizing looping matches

Click and drag the start or end point of the Animation Clip until the point appears on the graph where the property is green. Unity draws the graph in green where it is more likely that the clip can loop properly.



Position start and end points where the graph is green

When you let go of the mouse button, the graphs disappear but the indicators remain:



Green indicators show good matches for looping

2018–04–25 Page amended with limited editorial review

Curves

Leave feedback

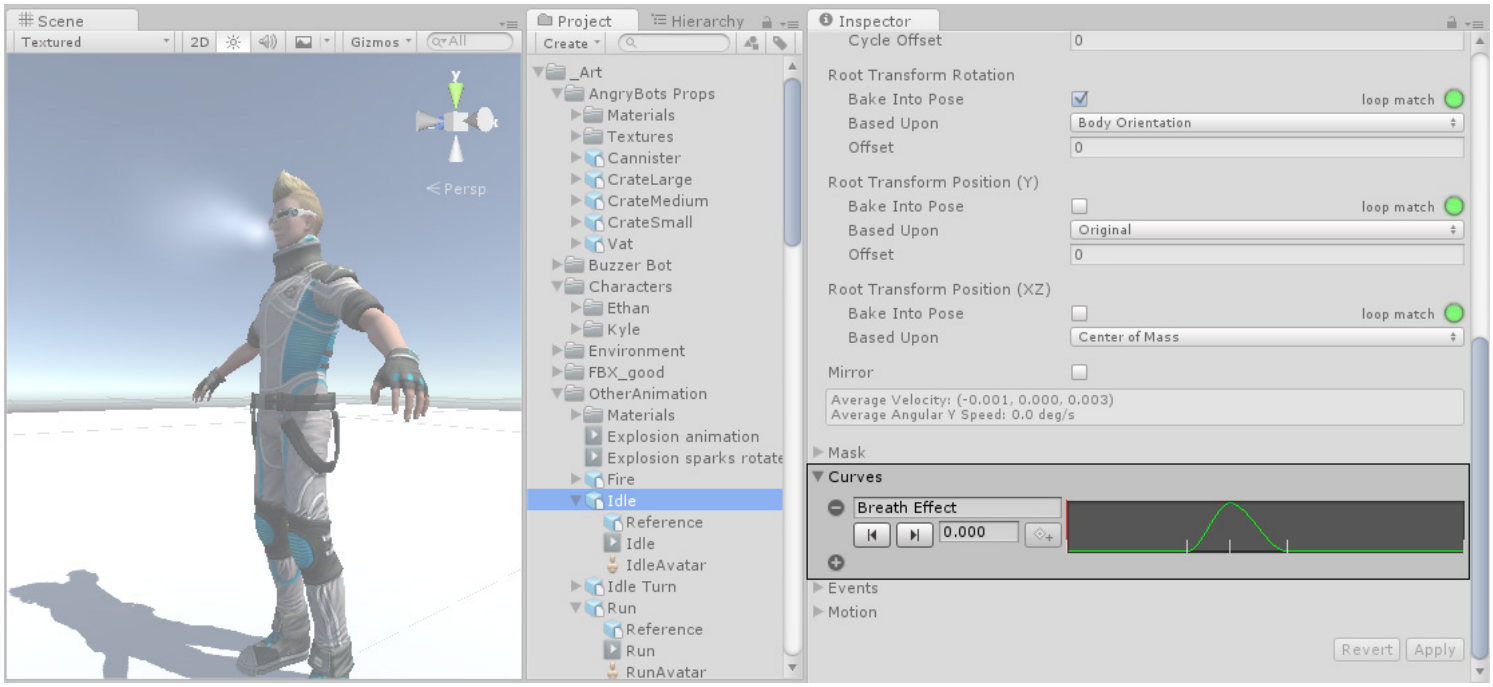
You can attach **animation curves** to imported **animation clips** in the Animatios tab.

You can use these curves to add additional animated data to an imported clip. You can use that data to animate the timings of other items based on the state of an animator. For example, in a game set in icy conditions, an extra animation curve could be used to control the emission rate of a **particle system** to show the player's condensing breath in the cold air.

To add a curve to an imported animation, expand the **Curves** section at the bottom of the Animation tab, and click the plus icon to add a new curve to the current animation clip:

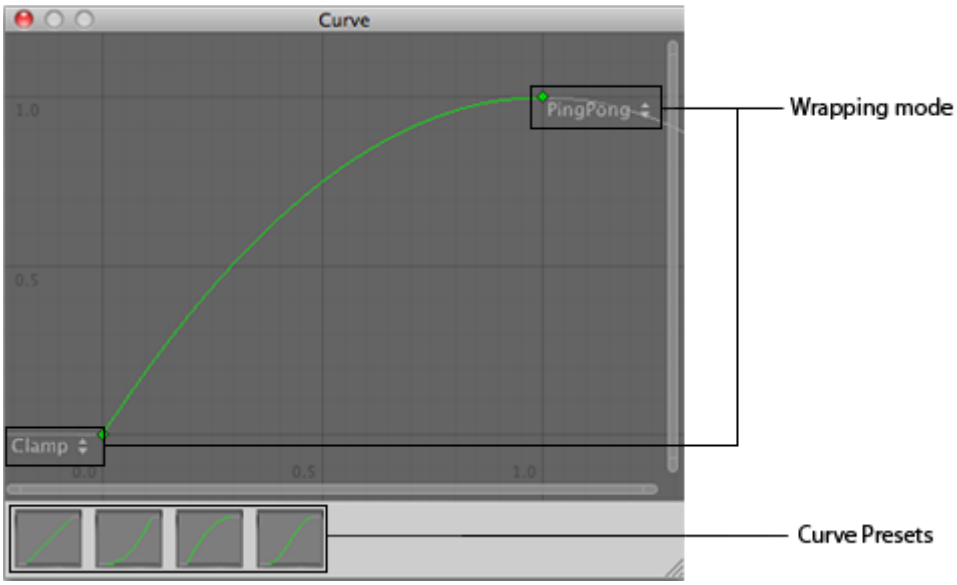
The expanded Curves section on the Animation tab

If your imported animation file is split into multiple animation clips, each clip can have its own custom curves.



The curves on an imported animation clip

The curve's X-axis represents *normalized time* and always ranges between 0.0 and 1.0 (corresponding to the beginning and the end of the animation clip respectively, regardless of its duration).



Unity Curve Editor

Double-clicking an animation curve brings up the standard Unity curve editor which you can use to add **keys** to the curve. Keys are points along the curve's **timeline** where it has a value explicitly set by the animator rather than just using an interpolated value. Keys are very useful for marking important points along the **timeline** of the animation. For example, with a walking animation, you might use keys to mark the points where the left foot is on the ground, then both feet on the ground, right foot on the ground, and so on. Once the keys are set up, you can move conveniently between key frames by pressing the **Previous Key Frame** and **Next Key Frame** buttons. This moves the vertical red line and showa the *normalized time* at the **keyframe**. The value you enter in the text box drives the value of the curve at that time.

Animation curves and animator controller parameters

If you have a curve with the same name as one of the parameters in the Animator Controller, then that parameter takes its value from the value of the curve at each point in the timeline. For example, if you make a call to GetFloat from a script, the returned value is equal to the value of the curve at the time the call is made. Note that at any given point in time, there might be multiple animation clips attempting to set the same parameter from the same controller. In that case, Unity blends the curve values from the multiple animation clips. If an animation has no curve for a particular parameter, then Unity blends with the default value for that parameter.

2018–04–25 Page amended with limited editorial review

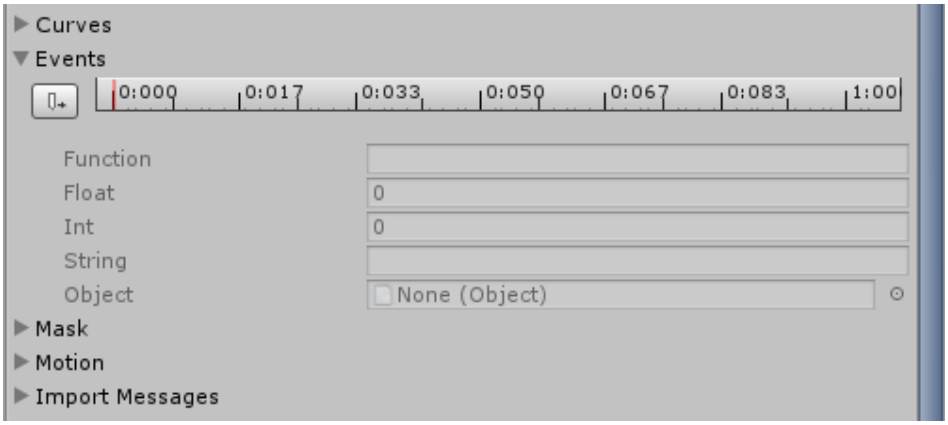
Events

Leave feedback

You can attach **animation events** to imported **animation clips** in the Animation tab.

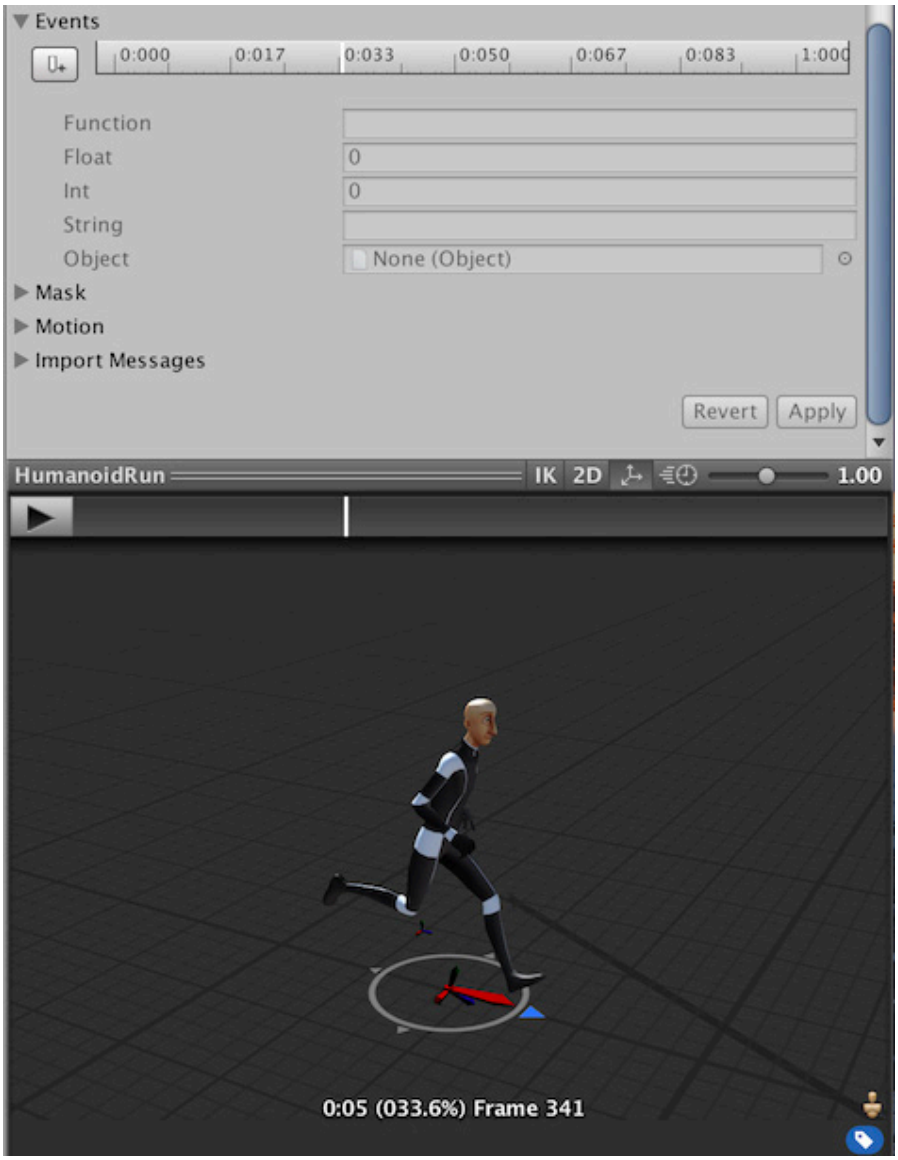
Events allow you to add additional data to an imported clip which determines when certain actions should occur in time with the animation. For example, for an animated character you might want to add events to walk and run cycles indicating when the footstep sounds should play.

To add an event to an imported animation, expand the Events section to reveal the events **timeline** for the imported animation clip:



The **Events** timeline, before any events have been added

To move the playback head to a different point in the timeline, use the timeline in the preview pane of the window:

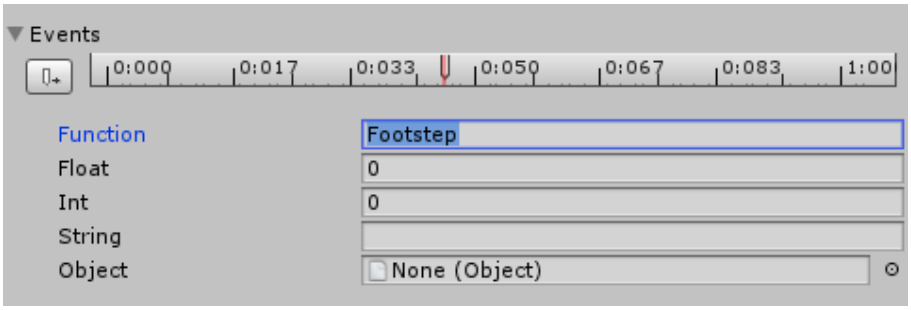


Clicking in the preview pane timeline allows you to control where you create your new event in the event timeline

Position the playback head at the point where you want to add an event, then click **Add Event**. A new event appears, indicated by a small white marker on the timeline. in the **Function** property, fill in the name of the function to call when the event is reached.

Make sure that any **GameObject** which uses this animation in its animator has a corresponding script attached that contains a function with a matching event name.

The example below demonstrates an event set up to call the `Footstep` function in a script attached to the Player GameObject. This could be used in combination with an AudioSource to play a footstep sound synchronised with the animation.



An event which calls the function “Footstep”

You can also choose to specify a parameter to be sent to the function called by the event. There are four different parameter types: **Float**, **Int**, **String** or **Object**.

By filling out a value in one of these fields, and implementing your function to accept a parameter of that type, you can have the value specified in the event passed through to your function in the script.

For example, you might want to pass a float value to specify how loud the footstep should be during different actions, such as quiet footstep events on a walking loop and loud footstep events on a running loop. You could also pass a reference to an effect **Prefab**, allowing your script to instantiate different effects at certain points during your animation.

2018–04–25 Page amended with limited editorial review

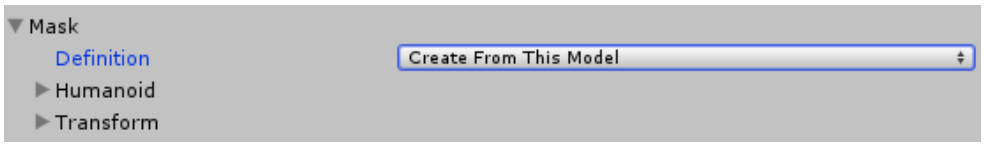
Mask

Leave feedback

Masking allows you to discard some of the animation data within a clip, allowing the clip to animate only parts of the object or character rather than the entire thing. For example, if you had a character with a throwing animation. If you wanted to be able to use the throwing animation in conjunction with various other body movements such as running, crouching and jumping, you could create a mask for the throwing animation limiting it to just the right arm, upper body and head. This portion of the animation can then be played in a layer over the top of the base running or jumping animations.

Masking can be applied to your build, making filesize and memory smaller. It also makes for faster processing speed because there is less animation data to blend at run-time. In some cases, import masking may not be suitable for your purposes. In that case you can use the layer settings of the **Animator Controller** to apply a mask at run-time. This page relates to masking in the import settings.

To apply a mask to an imported **animation clip**, expand the Mask heading to reveal the Mask options. When you open the menu, you'll see three options: Definition, Humanoid and Transform.



The Mask Definition, Humanoid and Transform options

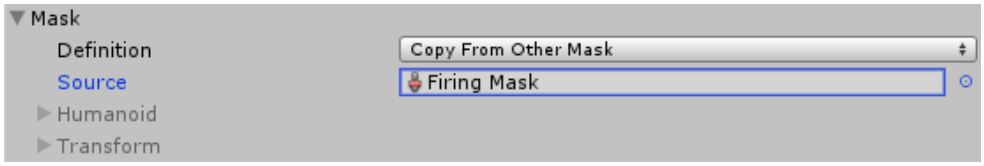
Definition

Allows you to specify whether you want to create a one-off mask in the **inspector** specially for this clip, or whether you want to use an existing mask **asset** from your project.

If you want to create a one-off mask just for this clip, choose / Create From This Model /.

If you are going to set up multiple clips with the same mask, you should select / Copy From Other Mask / and use a mask asset. This allows you to re-use a single mask definition for many clips.

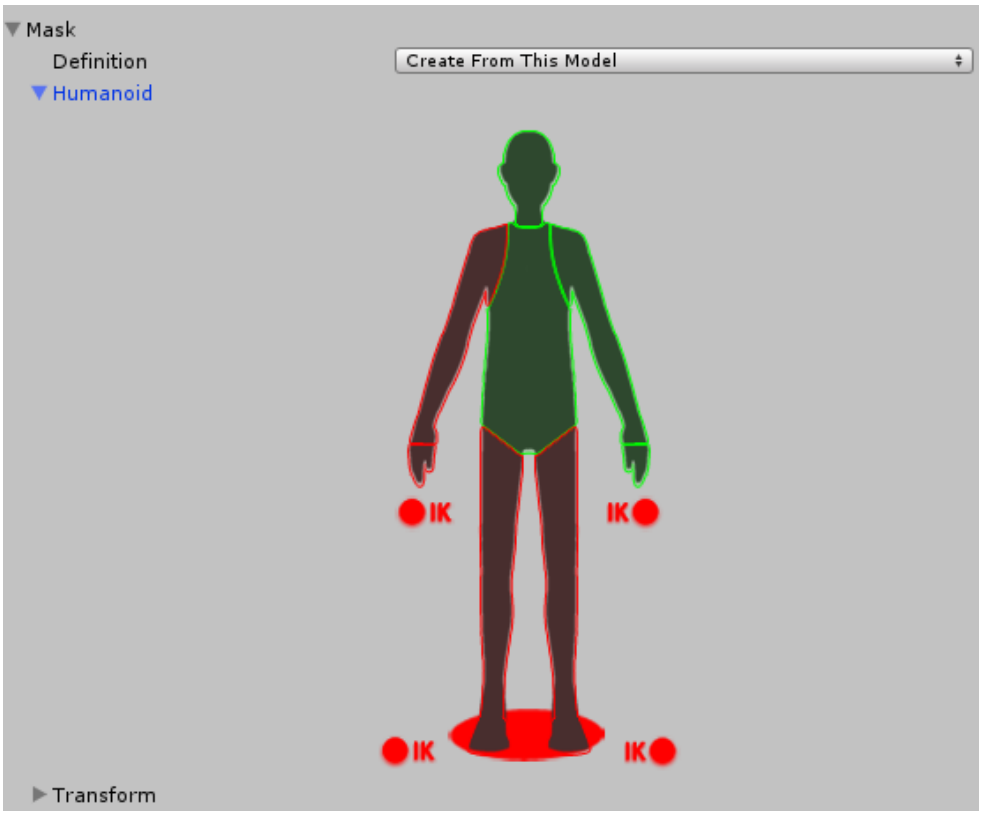
When Copy From Other Mask is selected, the Humanoid and Transform options are unavailable, since these relate to creating a one-off mask within the inspector for this clip.



Here, the Copy From Other option is selected, and a Mask asset has been assigned

Humanoid

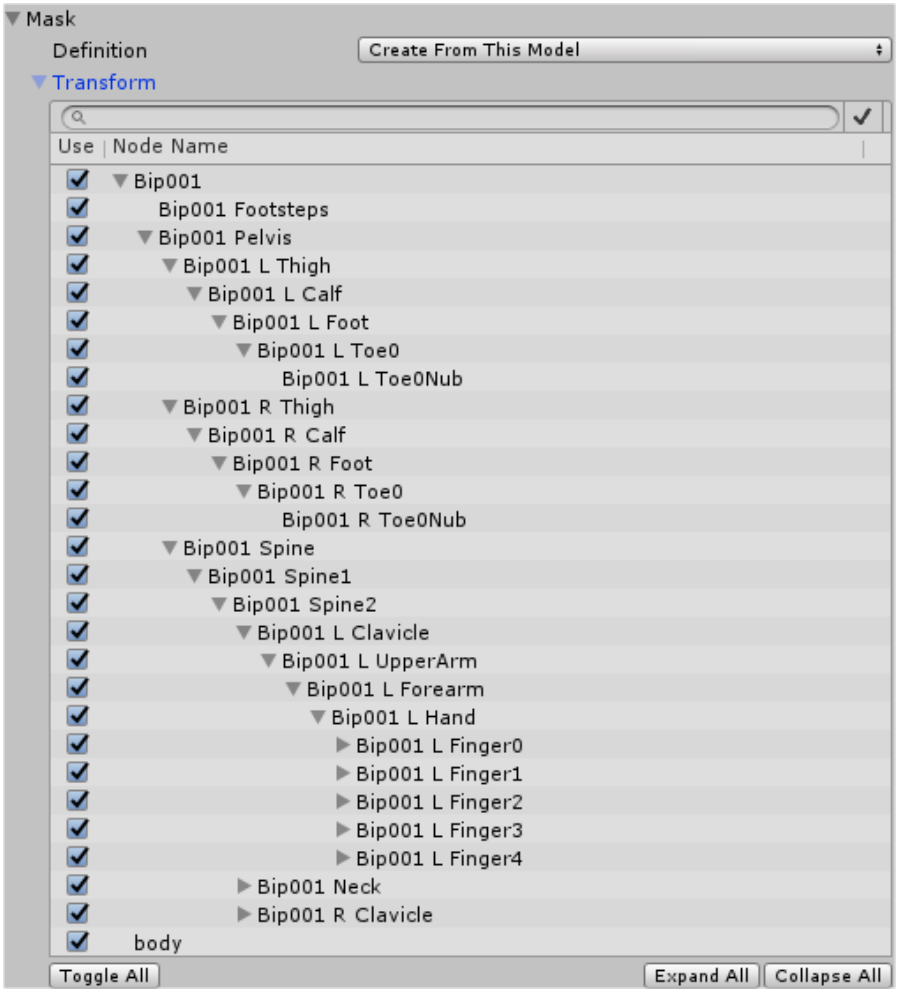
The Humanoid option gives you a quick way of defining a mask by selecting or deselecting the body parts of a human diagram. These can be used if the animation has been marked as humanoid and has a valid **avatar**.



The Humanoid mask selection option

Transform

This option allows you to specify a mask based on the individual bones or moving parts of the animation. This gives you finer control over the exact mask definition, and also allows you to apply masks to non-humanoid animation clips.



The Humanoid mask selection option

2018–04–25 Page amended with limited editorial review

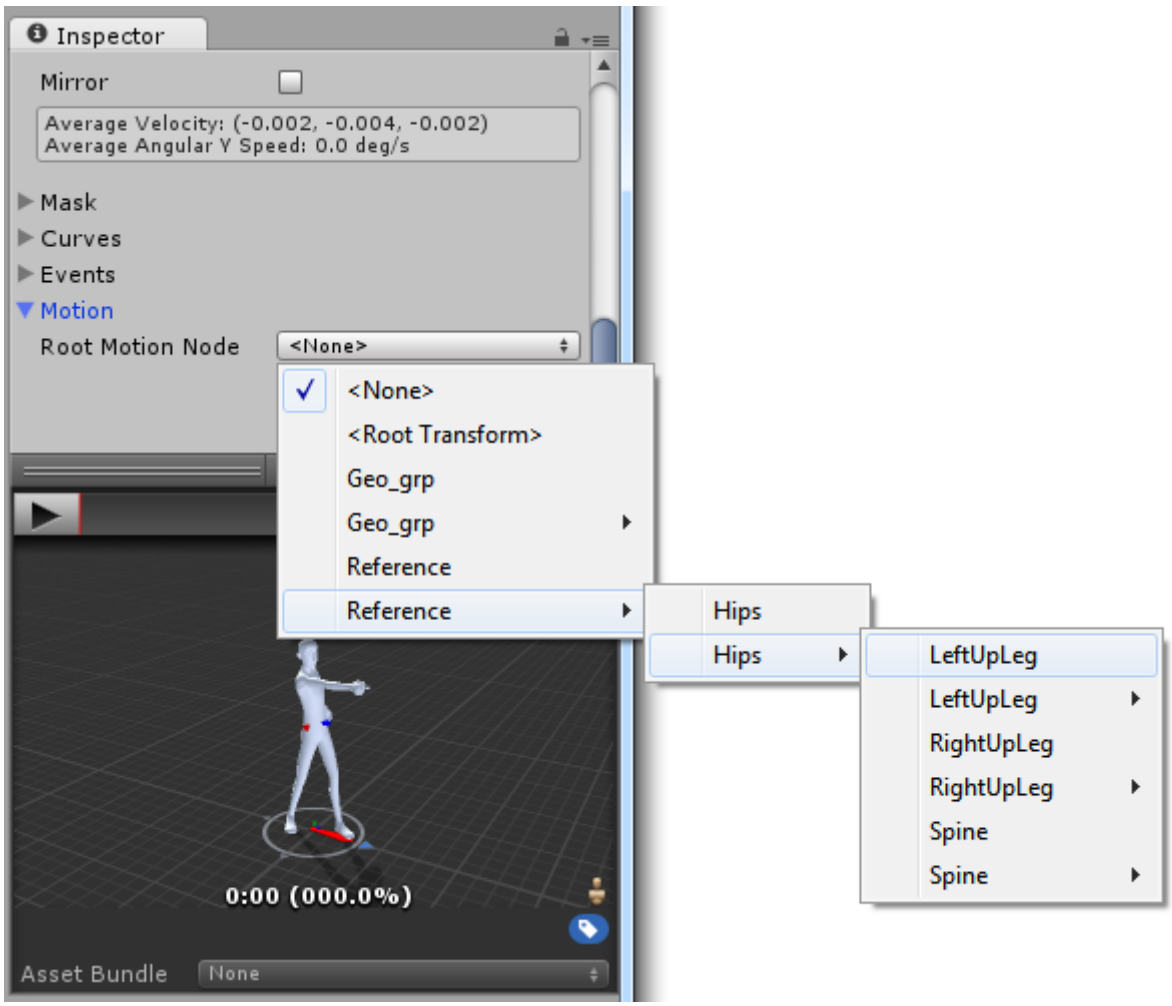
Motion

Leave feedback

When an imported **animation clip** contains **root motion**, Unity uses that motion to drive the movement and rotation of the **GameObject** that is playing the animation. Sometimes however it may be necessary to manually select a different specific node within the hierarchy of your animation file to act as the **root motion** node.

The Motion field within the Animation import settings allows you to use a hierarchical popup menu to select any node (Transform) within the hierarchy of the imported animation and use it as the root motion source. That object's animated position and rotation drive the animated position and rotation of the GameObject playing back the animation.

To select a root motion node for an animation, expand the Motion section to reveal the Root Motion Node menu. When you open the menu, any objects that are in the root of the imported file's hierarchy appear, including *None* and ***Root Transform*** . This may be your character's **mesh** objects, as well as its root bone name, and a submenu for each item that has child objects. Each submenu also contains the child object(s) itself, and further sub-menus if / those / objects have child objects.



Traversing the hierarchy of objects to select a root motion node

Once you select the Root motion node, the object's animation drives its motion.

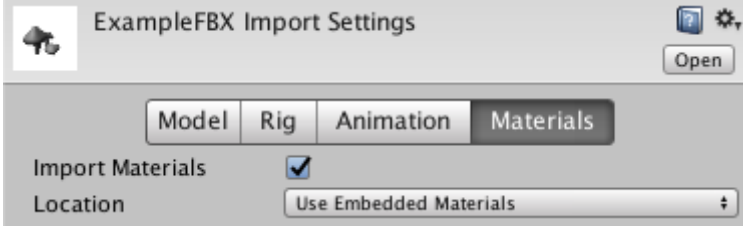
2018–04–25 Page amended with limited editorial review

Material tab

Leave feedback

You can use this tab to change how Unity deals with Materials and Textures when importing your model.

When Unity imports a Model without any Material assigned, it uses the Unity diffuse Material. If the Model has Materials, Unity imports them as sub-Assets. You can extract Embedded Textures into your Project using the **Extract Textures** button.

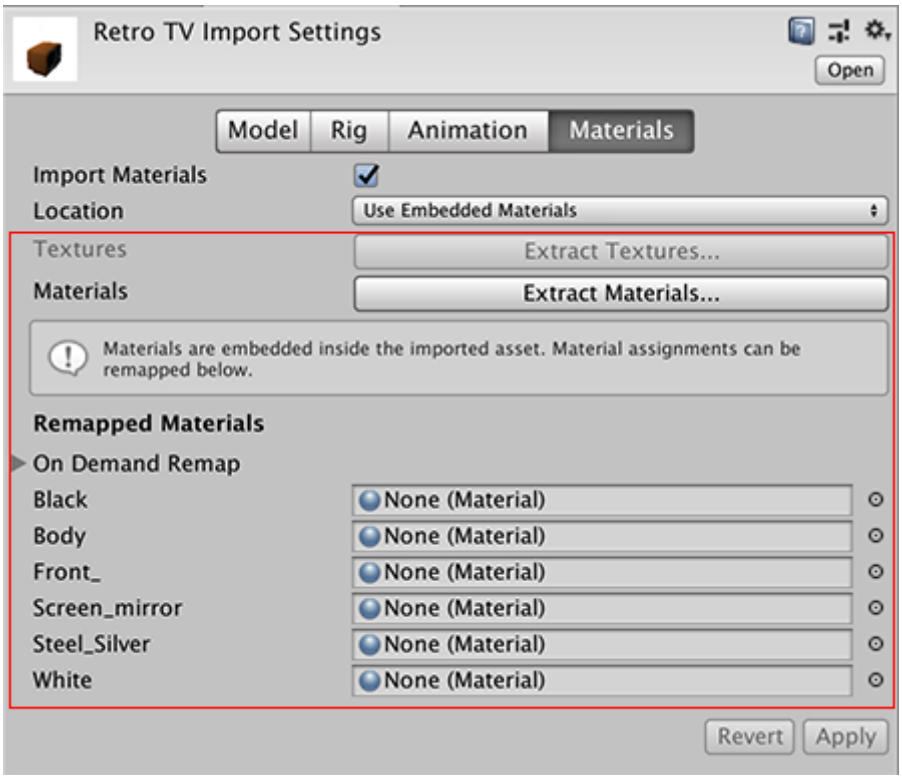


The Materials tab defines how Unity imports Materials and Textures

Property	Function
Import Materials	Enable the rest of the settings for importing Materials.
Location	Define how to access the Materials and Textures. Different properties are available depending on which of these options you choose.
Use Embedded Materials	Choose this option to keep the imported Materials inside the imported Asset. This is the default option from Unity 2017.2 onwards.
Use External Materials (Legacy)	Choose this option to extract imported Materials as external Assets. This is a Legacy way of handling Materials, and is intended for projects created with 2017.1 or previous versions of Unity.

Use Embedded Materials

When you choose Use **Embedded Materials** for the **Location** option, the following import options appear:



Import settings for **Use Embedded Materials**

Property	Function
Textures	Click the **Extract Textures** button to extract Textures that are embedded in your imported Asset. This is greyed out if there are no Textures to extract.
Materials	Click the **Extract Materials** button to extract Materials that are embedded in your imported Asset. This is greyed out if there are no materials to extract.
Remapped Materials	
On Demand Remap	These settings match the settings that appear in the **inspector** if you set the **Location** to Use External Materials (Legacy).
Naming	Define how Unity names the Materials.
By Base Texture Name	Use the name of the diffuse Texture of the imported Material to name the Material. When a diffuse Texture is not assigned to the Material, Unity uses the name of the imported Material.
From Model's Material	Use the name of the imported Material to name the Material.
Model Name + Model's Material	Use the name of the **model file** in combination with the name of the imported Material to name the Material.
Search	Define where Unity tries to locate existing Materials using the name defined by the **Naming** option.
Local Materials Folder	Find existing Materials in the "local" Materials folder only (that is, the *Materials* subfolder, which is the same folder as the model file).

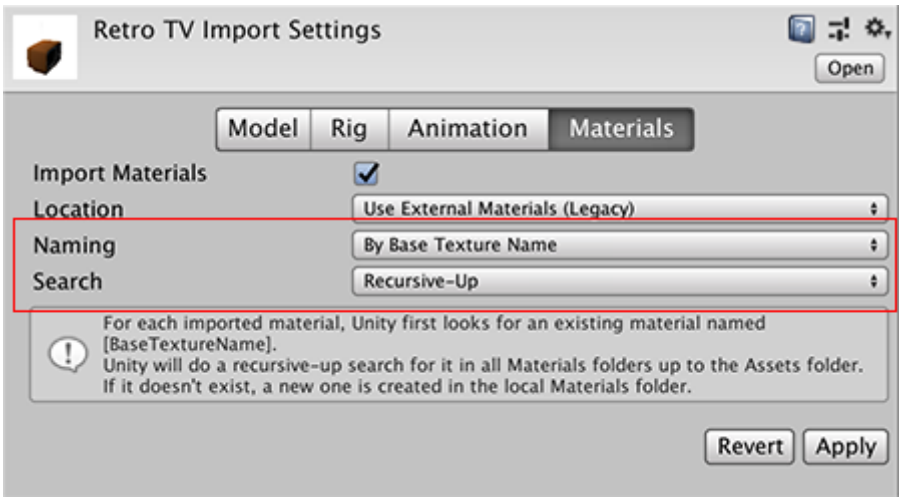
Property	Function
Recursive-Up	Find existing Materials in all Materials subfolders in all parent folders up to the Assets folder.
Project-Wide	Find existing Materials in all Unity project folders.
Search and Remap	Click this button to remap your imported Materials to existing Material Assets, using the same settings as the Legacy import option. Clicking this button does not extract the Materials from the Asset, and does not change anything if Unity cannot find any materials with the correct name.
List of imported materials	This list displays all imported Materials found in the Asset. You can remap each material to an existing Material Asset in your Project.

If you want to modify the properties of the Materials in Unity, you can extract them all at once using the **Extract Materials** button. If you want to extract them one by one, go to **Assets** > **Extract From Prefab**. When you extract Materials this way, they appear as references in the **Remapped Materials** list.

New imports or changes to the original Asset do not affect extracted Materials. If you want to re-import the Materials from the source Asset, you need to remove the references to the extracted Materials in the **Remapped Materials** list. To remove an item from the list, select it and press the Backspace key on your keyboard.

Use External Materials (Legacy)

When you choose **Use External Materials (Legacy)** for the **Location** option, the following import options appear:



Import settings for **Use External Materials (Legacy)**

Property	Function
Naming	Define how Unity names the Materials.
By Base Texture Name	Use the name of the diffuse Texture of the imported Material to name the Material. When a diffuse Texture is not assigned to the Material, Unity uses the name of the imported Material.
From Model's Material	Use the name of the imported Material to name the Material.

Property	Function
Model Name + Model's Material	Use the name of the model file in combination with the name of the imported Material to name the Material.
Search	Define where Unity tries to locate existing Materials using the name defined by the **Naming** option.
Local Materials Folder	Find existing Materials in the "local" Materials folder only (that is, the *Materials* subfolder, which is the same folder as the model file).
Recursive-Up	Find existing Materials in all Materials subfolders in all parent folders up to the Assets folder.
Project-Wide	Find existing Materials in all Unity project folders.

Before Unity version 2017.2, this was the default way of handling Materials.

2018–04–25 Page amended with limited editorial review

SketchUp Settings

Leave feedback

SketchUp is software that is commonly used for architecture modeling. Unity reads SketchUp files directly and supports the following SketchUp features:

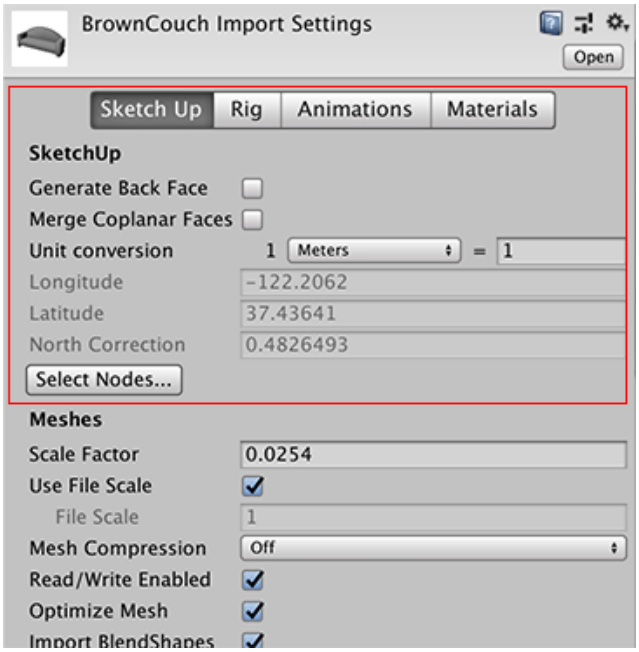
Textures and Materials, which Unity imports according to the settings on the Materials tab.
Component Definitions and Groups, which are converted into meshes, instanced as **GameObjects** which you can place in the **scene**.
Camera data for each scene in the file.
Tip: For information on how to export an FBX file from SketchUp, see Exporting from other applications.

Limitations

GIF Textures are not supported.
Only limited data from SketchUp Scenes are imported.
Unity does not support or import the following SketchUp features:
2D Components (Text, dimensions)
Animation settings
Attributes
Drawing Styles
Dynamic components
Layers
Lines
Section Planes
Shadow settings

SketchUp-specific Import Settings

To import a SketchUp file directly into Unity, drag it into the **Assets** folder using the Finder (MacOS) or the File Manager (Windows). When you click the Asset file inside Unity, the Model **Inspector** appears in a special **Sketch Up** tab:



SketchUp-specific properties in the Inspector window for importing the Model

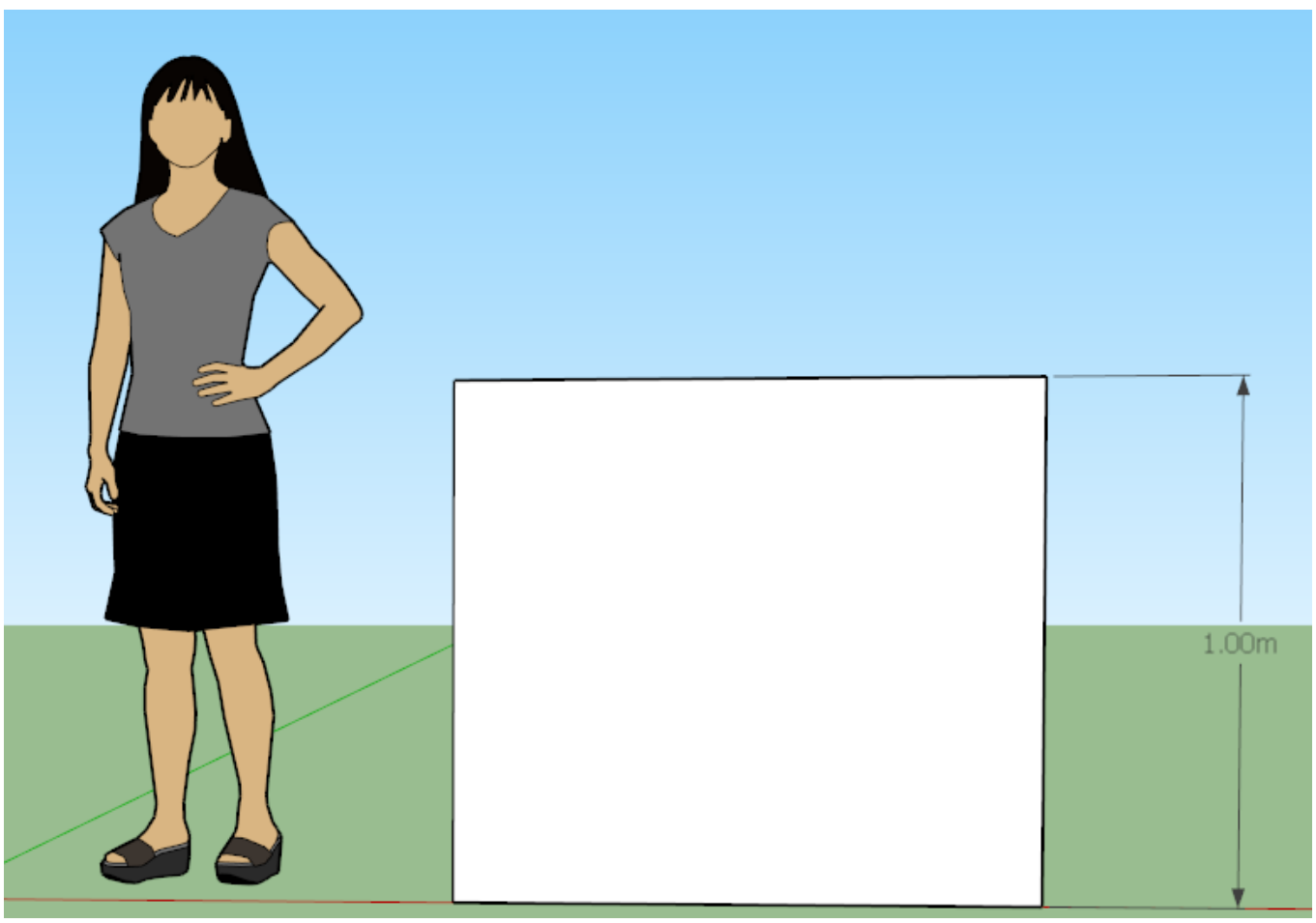
Property:	Function:
SketchUp	
Generate Back Face	Generate back-facing polygons in Unity. By default, Unity only imports the front-facing polygons to reduce polygon counts unless there is Material assigned to the back-facing polygons in SketchUp.

Property:	Function:
Merge Coplanar Faces	Merge coplanar faces when generating meshes in Unity.
Unit Conversion	Length measurement to unit conversion.
Unit drop-down box	Choose the unit to use for the conversion. Defaults to *Meters*.
Value to convert	This value determines how the **Scale Factor** is calculated (see Unit conversion below).
Longitude	Read-only value from the *Geo Coordinate* system, used to identify a position on a geographic system.
Latitude	Read-only value from the *Geo Coordinate* system, used to identify a position on a geographic system.
North Correction	Read-only value from the *Geo Coordinate* system, used to describe the angle needed to rotate North to the Z axis.
Select Nodes	Open a window where you can specify which nodes to import. A node represents an Entity, Group, or Component Instance in SketchUp. For example, if you have one file containing several couches, you can select the one you want to import. For more information, see Selecting SketchUpNodes below.

The rest of the options on the Inspector window are the regular FBX Model import options that are available for any 3D modeling application.

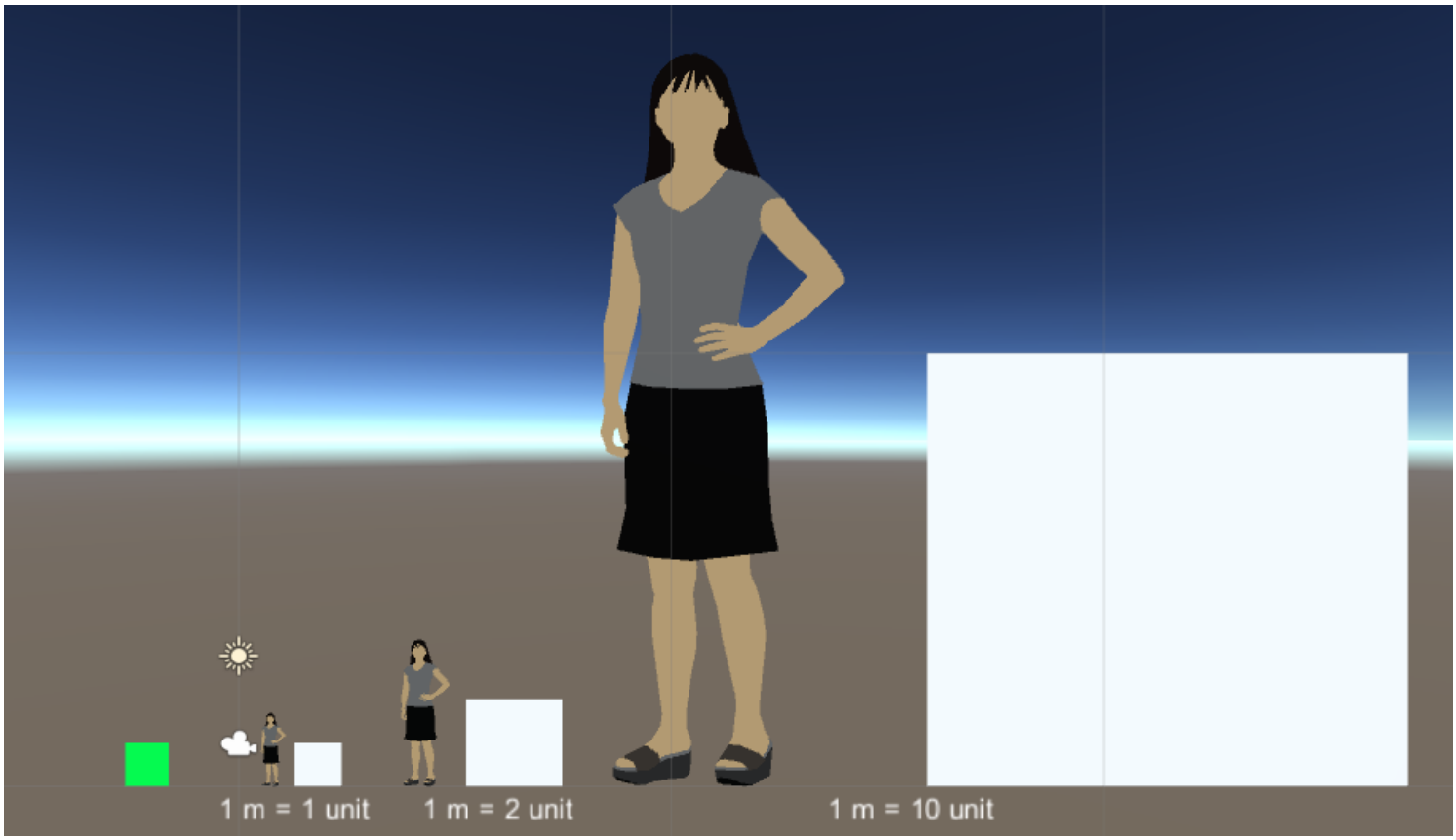
Unit conversion

By default, Unity scales SketchUp models to 1 meter (0.0254 inches) to 1 unit length.



SketchUp file with a cube set to 1m in height

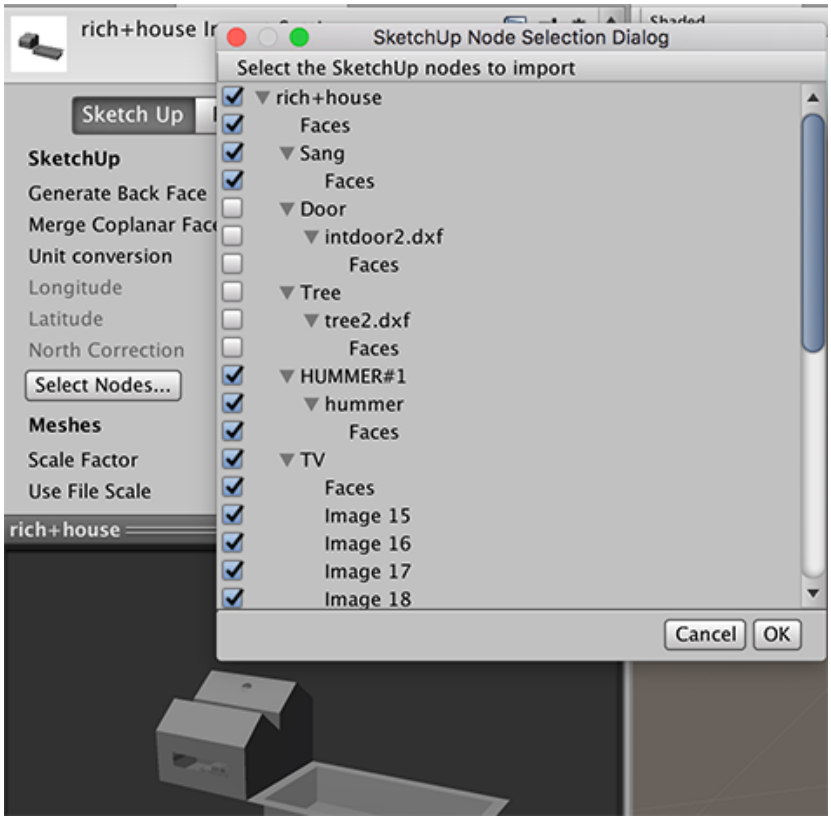
Changing the default **Unit Conversion** values affects the scale of the imported file:



The green square is placed as reference where the size of the square is set to 1x1 unit length.

Selecting SketchUp Nodes

Unity supports the visibility setting in the SketchUp file for each node. If a node is hidden in the SketchUp file, Unity does not import the node by default. However, you can override this behavior by clicking the **Select Nodes** button to display the SketchUp node hierarchy in the SketchUp Node Selection Dialog window.



SketchUp Node Selection Dialog window

Each group and component instance in the file appears in the hierarchy as a node, which you can select or deselect. When you are finished selecting the nodes to include, click the OK button to import only the checked nodes.

2018–04–25 Page amended with limited editorial review

Model file formats

Leave feedback

Unity supports importing Meshes and animation from two different types of files:

Exported 3D file formats, such as .fbx or .obj. You can export files from 3D modeling software in generic formats that can be imported and edited by a wide variety of different software.

Proprietary 3D or DCC (Digital Content Creation) application files, such as .max and .blend file formats from Autodesk® 3ds Max® or Blender, for example. You can only edit proprietary files in the software that created them. Proprietary files are generally not directly editable by other software without first being converted and imported. An exception to this is SketchUp .skp files, which both SketchUp and Unity can read.

Unity can import and use both types of files, and each come with their own advantages and disadvantages.

Exported 3D files

Unity can read .fbx, .dae (Collada), .3ds, .dxf, .obj, and .skp files. For information about exporting 3D files, see Exporting from other applications or read the documentation for your 3D modeling software.

Advantages:

You can import only the parts of the model you need instead of importing the whole model into Unity.
Exported generic files are often smaller than the proprietary equivalent.
Using exported generic files encourages a modular approach (for example, using different components for collision types or interactivity).
You can import these files from software that Unity does not directly support.
You can re-import exported 3D files (.fbx, .obj) into 3D modeling software after exporting, to ensure that all of the information has been exported correctly.

Disadvantages:

You must re-import models manually if the original file changes.
You need to keep track of versions between the source file and the files imported into Unity.

Proprietary 3D application files

Unity can import proprietary files from the following 3D modeling software:

Autodesk® 3ds Max®
Autodesk® Maya®
Blender
Cinema4D
Modo
LightWave
Cheetah3D

Warning: Unity converts proprietary files to .fbx files as part of the import process. However, it is recommended that you export to FBX instead of directly saving your application files in the Project. It is not recommended to use native file formats directly in production.

Advantages:

Unity automatically re-imports the file if the original model changes.
This is initially simple; however it can become more complex later in development.

Disadvantages:

A licensed copy of the software used must be installed on each machine that uses the Unity Project.
Software versions should be the same on each machine using the Unity Project. Using a different software version can cause errors or unexpected behavior when importing 3D models.
Files can become bloated with unnecessary data.
Big files can slow down Unity Project imports or **Asset** re-imports, because you have to run the 3D modeling software you use as a background process when you import the model into Unity.
Unity exports proprietary files to .fbx internally, during the import process. This makes it difficult to verify the .fbx data and troubleshoot problems.

Note: Assets saved as .ma, .mb, .max, .c4d, or .blend files fail to import unless you have the corresponding 3D modeling software installed on your computer. This means that everybody working on your Unity Project must have the correct software installed. For example, if you use the Autodesk® Maya LT™ license to create `ExampleModel.mb` and copy it into your Project, anyone else opening that Project also needs to have Autodesk® Maya LT™ installed on their computer too.

2018–04–25 Page amended with limited editorial review

Limitations when importing from other applications

Leave feedback

When Unity imports a proprietary file, it launches the 3D modeling software in the background. Unity then communicates with that proprietary software to convert the native file into a format Unity can read.

The first time you import a proprietary file into Unity, the 3D modeling software has to launch in a command-line process. This can take a while, but subsequent imports are very quick.

Warning: It is recommended that you export to FBX instead of directly saving your application files in the Project. It is not recommended to use native file formats directly in production.

Requirements

You need to have the 3D modeling software installed to import proprietary files directly into Unity. If you don't have the software installed, use the FBX format instead. For more information about importing FBX files, see Model Import Settings window.

Application-specific issues

You import files in the same way, regardless of whether they are generic or proprietary files. However, there are some differences between which features are supported. For more information on the limitations with a specific 3D application, see:

Importing objects from Autodesk® Maya®
Importing objects from Cinema 4D
Importing objects from Autodesk® 3ds Max®
Importing objects from Cheetah3D
Importing objects from Modo
Importing objects from LightWave
Importing objects from Blender
SketchUp Settings

Importing objects from Autodesk® Maya®

Unity imports Autodesk® Maya® files (*.mb* and *.ma*) through the FBX format, supporting the following:

All nodes with position, rotation and scale; pivot points and names are also imported
Meshes with vertex colors, normals and up to 2 UV sets
Materials with texture and diffuse color; multiple materials per mesh
Animation
Joints
Blendshapes
Lights and Cameras
Visibilty
Custom property animation
Tip: For information on how to export an FBX file from Autodesk® Maya®, see Exporting from other applications.

Limitations

Unity does not support Autodesk® Maya®'s *Rotate Axis* (pre-rotation).

Joint limitations include:

Joint Orient (joint only post-rotation)
Segment Scale Compensate (joint only option)
Unity imports and supports any *Rotate Order* you specify in Autodesk® Maya®; however, once imported, you cannot change that order again inside Unity. If you import a Model that uses a different rotation order from Unity's, Unity displays that rotation order

in the **Inspector** beside the **Rotation** property.

Tips and troubleshooting

Keep your **scene** simple: only export the objects you need in Unity when exporting.
Unity only supports polygons, so convert any patches or NURBS surfaces into polygons before exporting; see Autodesk® Maya® documentation for instructions.
If your model did not export correctly, the node history in Autodesk® Maya® might be causing a problem. In Autodesk® Maya®, select **Edit** > **Delete by Type** > **Non-Deformer History** and then re-export the model.
The Autodesk® Maya® FBX Exporter bakes un-supported complex animations constraints, such as Set Driven Keys, in order to import the animation into Unity properly. If you are using Set Driven Keys in Autodesk® Maya®, make sure to set keys on your drivers in order for the animation to be baked properly. For more information, see the Autodesk® Maya® documentation on **Keyframe** Animation.
In Autodesk® Maya®, the visibility value is present on each shape but can't be animated and is not exported to the FBX file. Always set the visibility value on a node and not on a shape.

Importing objects from Cinema 4D

Unity imports Cinema 4D files (*.c4d*) through the FBX format, supporting the following:

All objects with position, rotation and scale; pivot points and names are also imported
Meshes with UVs and normals
Materials with texture and diffuse color; multiple materials per mesh
Animations FK (IK needs to be baked manually)
Bone-based animations
Tip: For information on how to export an FBX file from Cinema 4D, see Exporting from other applications.

Limitations

Unity does not import Cinema 4D's Point Level Animations (PLA). Use bone-based animations instead.

Cinema 4D does not export visibility inheritance. Set the Renderer to 'Default' or 'Off' in Cinema 4D to avoid any difference in the visibility animation between Cinema4D and Unity.

Importing Objects From Autodesk® 3ds Max®

Unity imports Autodesk® 3ds Max® files (*.max*) through the FBX format, supporting the following:

All nodes with position, rotation and scale; pivot points and names are also imported
Meshes with vertex colors, normals and one or more UV sets
Materials with diffuse texture and color. Multiple materials per mesh
Animations
Bone-based animations
Morphing (Blendshapes)
Visibility
Note: Saving a Autodesk® 3ds Max® file (.max) or exporting a generic 3D file type (.fbx) each has advantages and disadvantages see class-Mesh.

Tip: For information on how to export an FBX file from Autodesk® 3ds Max®, see Exporting from other applications.

Importing objects from Cheetah3D

Unity imports Cheetah3D files (*.jas*) through the FBX format, supporting the following:

All nodes with position, rotation and scale; pivot points and names are also imported
Meshes with vertices, polygons, triangles, UVs, and normals
Animations

Materials with diffuse color and textures
Tip: For information on how to export an FBX file from Cheetah3D, see Exporting from other applications.

Importing objects from Modo

Unity imports Modo files (*.lxo*) through the FBX format, supporting the following:

All nodes with position, rotation and scale; pivot points and names are also imported
Meshes with vertices, normals and UVs.
Materials with Texture and diffuse color; multiple Materials per mesh
Animations
To get started, save your **.lxo** file in your Project's **Assets** folder. In Unity, the file appears in the **Project View**.

Unity re-imports the Asset when it detects a change in the .lxo file.

Tip: For information on how to export an FBX file from Modo, see Exporting from other applications.

Importing objects from Lightwave

Unity imports Lightwave files through the FBX format, supporting the following:

All nodes with position, rotation and scale; pivot points and names are also imported
Meshes with up to 2 UV channels
Normals
Materials with Texture and diffuse color; multiple materials per mesh
Animations
Bone-based animations
You can also configure the Lightwave AppLink plug-in which automatically saves the FBX export settings you use the first time you import your Lightwave scene file into Unity. For more information, see the Lightwave Unity Interchange documentation.

Tip: For information on how to export an FBX file from Lightwave file, see Exporting from other applications.

Limitations

Bake your Lightwave-specific materials as textures so that Unity can read them. For information on doing this using a non-destructive pipeline, see Node system in Lightwave.

Unity does not support splines or patches. Convert all splines and patches to polygons before saving and exporting to Unity. For more information, see Lightwave documentation.

Importing objects from Blender

Unity imports Blender (*.blend*) files through the FBX format, supporting the following:

All nodes with position, rotation and scale; pivot points and names are also imported
Meshes with vertices, polygons, triangles, UVs, and normals
Bones
Skinned Meshes
Animations
For information on how to optimize importing your Blender file into Unity, see Exporting from other applications.

Limitations

Textures and diffuse color are not assigned automatically. You can manually assign them by dragging the texture onto the **mesh** in the **Scene View** in Unity.

Blender does not export the visibility value inside animations in the FBX file.

2018–04–25 Page amended with limited editorial review

Exporting from other applications

Leave feedback

Unity supports FBX files which you can export from many 3D modeling applications. Use these guidelines to help ensure the best results:

Select what you want to export inside your 3D modeling application.
Prepare what you need to include inside your 3D modeling application.
Check the FBX settings before exporting.
Verify and import the FBX file into Unity.

Note: In addition to these generation guidelines about exporting from 3d modeling applications, some 3d modeling applications have more specific information under these sections:

Autodesk® Maya®
MAXON Cinema 4D
Autodesk® 3ds Max®
NewTek LightWave

Selecting what to export

Think about what you want to export: some **scene** objects are essential but others may be unnecessary. Applications often let you export **selected objects** or a **whole scene**. You can optimize the data in Unity by keeping only the essential objects.

If you choose to export only specific objects in your scene, you can:

Export only the **selected objects** if your application supports it.
Remove unwanted data from your scene and export the **whole scene**.
Make a preset or a custom scene exporter to automate or simplify the selection export process.

Preparing what you need to include

Prepare your **Assets** for export, using the following considerations:

Object:	Preparations:
Meshes	All NURBS, NURMS, splines, patches, and subdiv surfaces must be converted to polygons (triangulation or quadrangulation)
Animation	Select the correct rig. Check the frame rate. Check the animation length.
Blend Shapes or **Morphing**	Make sure your Blend Shapes or Morph targets are assigned. Set up the export **Mesh** appropriately.
Bake deformers	Make sure that you bake deformers onto your Model before exporting to FBX. For example, if you are exporting a complex rig from Maya, you can bake the deformation onto skin weights before you export the Model to FBX.
Textures	Make sure your textures are either sourced from your Unity Project or copy them into a folder called `textures` inside your Unity Project. **Note:** We don't recommend embedding textures in the FBX file using the **Embed Media** option. You must extract textures before using them, so embedding them just bloats your project unnecessarily and slows the import process.
Smoothing	Verify any smoothing groups and smooth Mesh settings. **Important:** Importing blendshape normals requires having smoothing groups in the FBX file.

Setting the FBX export options

Check your FBX export settings:

Check each setting in the export dialogue of your 3D modeling application so that you know what to match up the FBX import settings in Unity.
Select whether to export Animation, Deformations, Skins, Morphs according to your needs.
Nodes, markers and their transforms can be exported to Unity.
Select any Cameras, Lights, and Visibility settings you want to import into Unity.
Use the Latest Version of FBX where possible. Autodesk regularly updates their FBX installer.
Make sure you use the same FBX version to import files into Unity as you used to export them from your 3D modeling application. Using different versions of the software can provide unexpected results.

Verifying and importing into Unity

Before importing your FBX file into Unity:

Verify the size of your exported file. Perform a sanity check on the file size (e.g. >10kb?).

Re-import your FBX file into a new scene back into the 3D modeling software you used to generate it. Check to make sure it is what you expected.

To import your file into Unity, follow the instructions for Importing, keeping in mind how you set the export options in your 3D modeling software.

Scaling factors

Unity's physics and lighting systems expect **1** meter in the game world to be **1** unit in the imported file.

The defaults for different 3D packages are as follows:

.fbx, .max, .jas, .c4d = **0.01**
.mb, .ma, .lxo, .dxf, .blend, .dae = **1**
.3ds = **0.1**
When importing **Model files** into Unity from a 3D modeling application with a different scaling factor, you can convert the file units to use the Unity scale by enabling the **Convert Units** option.

Export settings in specific 3D modeling applications

Autodesk® Maya®

You can use the FBX Export options to pick what to export in your FBX file.

Exporting BlendShapes (morphing)

When exporting BlendShapes (Morphing) from Maya, follow these guidelines:

Apply the blend shape to the export mesh with its targets in order.
If you require Maya animation **keyframes**, you can animate **keyframes** on the blend shape node.
Enable the **Animation** > **Deformed Models** > **Blend Shapes** FBX Export option in Maya before exporting the mesh.
If you also want to export skin deformation, enable the **Animation** > **Deformed Models** > **Skins** FBX Export option in Maya before exporting the mesh.
When you've imported the file into Unity, select it in the **Project** view and enable **Bake Animations** in the Animations tab of the Model Importer.

Exporting complex deformation

You can create very complex character rigs in Maya. For performance reasons, Unity only supports linear blend **skinning** with four influences per vertex. If your character uses more than four influences, when you import the Model into Unity, the animation may appear choppy or distorted. Or if you are using deformation other than clusters, the animation may be completely absent.

To solve this problem, bake the deformation joints before exporting your Model from Maya, using the *Bake Deformer* tool (from Maya's top menu: **Skin** > **Bake Deformers to Skin Weights**).

For more information, see Bake deformers on a character in the Autodesk documentation.

Cinema 4D

Animated characters using IK

If you are using IK to animate your characters in Cinema 4D, you have to bake the IK before exporting using the **Plugins > Mocca > Cappucino** menu in Cinema 4D. If you don't bake your IK prior to importing into Unity, you only get animated locators and no animated bones.

Maximizing import speed

You can speed up file import into Unity by turning off the **Embed Textures** preference in Cinema 4D before you export. Check the Cinema 4D documentation for instructions.

Autodesk® 3ds Max®

You can use the FBX Export options to pick what to export in your FBX file.

When exporting from 3ds Max, there are some extra considerations when dealing with the following:

Exporting quads
Bone-based Animations
Morph targets (Blendshapes)
UV sets for Lightmapping

Exporting quads

3ds Max's editable Mesh always exports triangles. The Editable Poly retains **quads** and N-gons on import. So if you want to import **quads** into Unity, you have to use an Editable Poly in 3ds Max.

Bone-based Animations

Follow these guidelines when you want to export bone-based animations in 3ds Max:

After setting up the bone structure and the animations (using FK or IK), select all bones and/or IK solvers.
Select **Motion** > **Motion Paths** > **Collapse**. Unity makes a key filter, so the amount of keys you export is irrelevant.
Click the OK button on the FBX Exporter window.
Copy the FBX file into the Assets folder.
Open Unity and reassign the Texture to the Material in the root bone.
When exporting a bone hierarchy with Mesh and animations from 3ds Max to Unity, the exported **GameObject** hierarchy corresponds to the hierarchy you can see in the 3ds Max Schematic View. However, Unity uses a **GameObject** as the new root, containing the animations, and places the Mesh and material information in the root bone.

If you prefer to keep animation and Mesh information in the same Unity GameObject, parent the Mesh node to a bone in the bone hierarchy in 3ds Max before exporting.

Morph targets (Blendshapes)

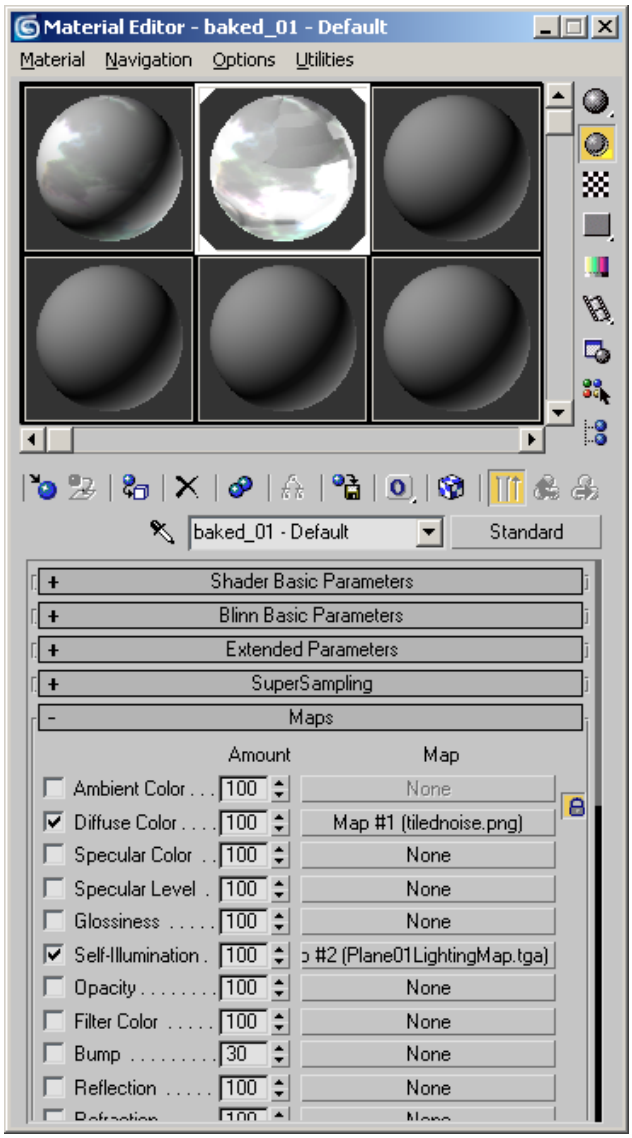
Follow these guidelines when you want to export Morph targets in 3ds Max:

Apply the Morpher Modifier to the export Mesh with appropriate morph targets set up in the Channel List.
If you require 3ds Max animation keyframes, you can animate keyframes on the Mesh/modifier.
Before exporting the Mesh, eEnable the **Animation** > **Deformed Models** > **Blend Shapes** FBX Export option in 3ds Max.
If you also want to export skin deformation, enable the **Animation** > **Deformed Models** > **Skins** FBX Export option in 3ds Max before exporting the Mesh.

UV sets for Lightmapping

Unity has a built-in lightmapper, but you can also create **lightmaps** using texture baking (**Render To Texture** on the Rendering menu) feature and automatic unwrapping functionality in 3ds Max.

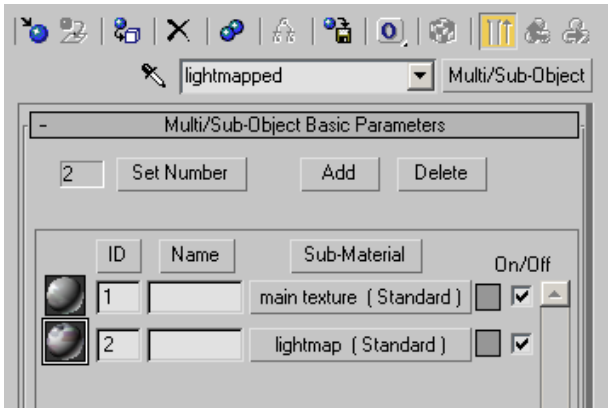
Usually one UV set is used for main texture and/or normal maps, and another UV set is used for the lightmap Texture. For both UV sets to come through properly, set the Material in 3ds Max to Standard and set up both Diffuse (for the main Texture)and Self-Illumination (for the lightmap) map slots:



Material setup for Lightmapping in 3ds Max, using self-illumination map

Note: If the object uses a Shell type of Material, Autodesk's FBX exporter does **not export UVs correctly**.

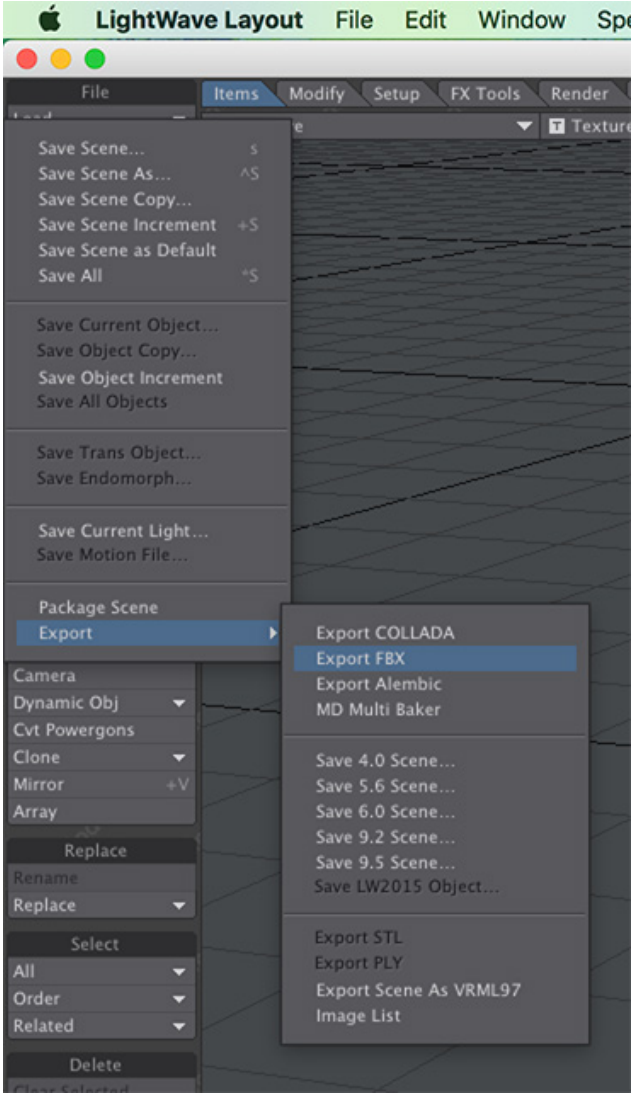
Alternatively, you can use Multi/Sub Object material type and setup two sub-materials, using the main texture and the lightmap in their diffuse map slots, as shown below. However, if the faces in your Model use different sub-material IDs, this results in multiple materials being imported, which is not optimal for performance.



Alternate Material setup for Lightmapping in 3ds Max, using multi/sub object material

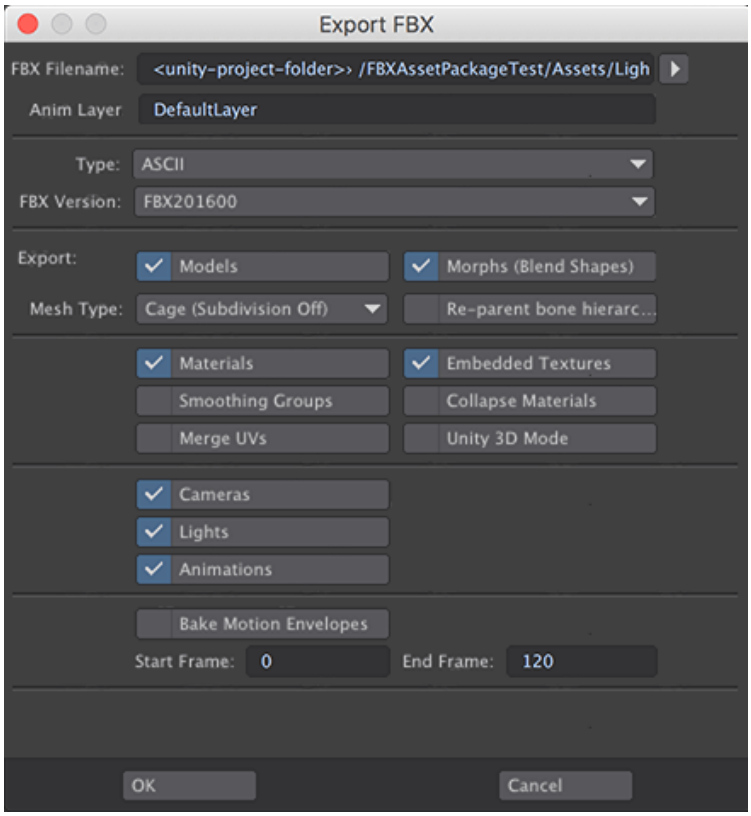
LightWave

You can access the FBX export settings window inside LightWave by selecting **Save** > **Export** > **Export FBX** from the **File** toolbar in LightWave Layout:



Accessing FBX Export options in LightWave Layout

The Export FBX window appears.



LightWave FBX Export Settings

Property:	Function:
FBX Filename	Set the name and location of the FBX file. Use a location under the Unity `Assets` folder.
Anim Layer	Name of the **animation layer** to use.
Type	Use **Binary** to reduce filesize or **ASCII** for a text-editable FBX file.
FBX Version	Select the most recent FBX version from the list, making sure that it matches the version that Unity is using.
Export	
Models	Export all models in the scene
Morphs (Blend Shapes)	Export all BlendShapes in the scene
Mesh type	
Cage (Subdivision Off)	Export the object without any subdivision applied to it
Subdivision	Subdivide the mesh when it is exported
Re-parent bone hierarchy	Create a null to act as the new parent of the bone hierarchy. When exporting a rig from Layout with the bone hierarchy parented to the mesh, the actual movement of the deformed mesh is twice what it should be. Enabling this new parent keeps the mesh in place.
Materials	Convert LightWave's standard Surface channels and image maps. This does not include procedural textures and nodes.
Embedded Textures	Save embedded Textures as image maps included directly in the FBX file instead of in a separate image directory. This creates much larger, but self-contained, FBX files.
Smoothing Groups	Convert LightWave's normals into smoothing groups.

Property:	Function:
Collapse Materials	Collapse surfaces with identical Material names, exporting the Materials separately. However, if both Material names and all surface parameters match, the two Materials are always merged, regardless of this setting.
Merge UVs	Collapse multiple UV maps into a single map per object.
Unity 3D Mode	Correct rotation errors caused by converting between coordinate systems across LightWave, FBX, and Unity. Both LightWave and Unity use *left-handed* coordinate systems but FBX is *right-handed*. When LightWave exports to FBX, it converts to right-handed coordinates along the Z axis. When Unity imports the FBX file, it converts back to left-handed coordinates along the X axis, which results in an apparent 180 degree rotation of the scene. Using the setting means that when you go into Unity and look down the Z axis, the imported FBX looks exactly the same as it does inside LightWave.
Cameras	Export all cameras in the scene.
Lights	Export all lights in the scene.
Animations	Export simple animations based on movement, rotation or scaling without baking. Character animation or other animation using IK or dynamics should still be baked, using **Bake Motion Envelopes**.
Bake Motion Envelopes	Set an arbitrary start and end point for baking, in case there are setup frames you do not wish to capture. Only available if **Animations** is checked.
Start Frame and **End Frame**	Export data only inside this timeframe.
Scale Scene	Set a scale for your scene to match the Unity file scale value.

2018–09–26 Page amended with limited editorial review

Humanoid Asset preparation

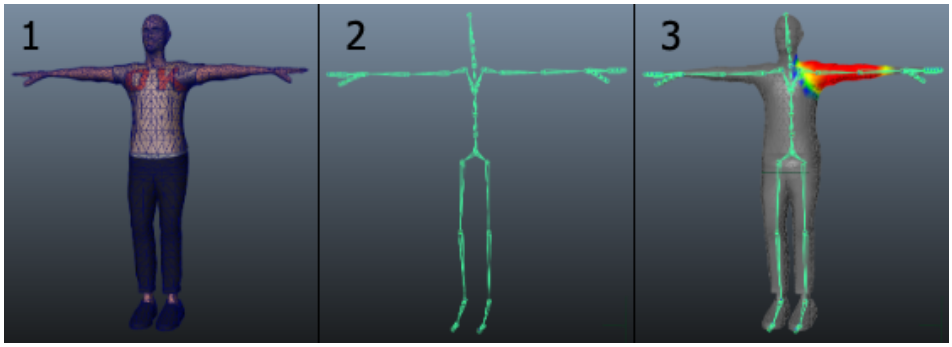
Leave feedback

In order to take full advantage of Unity's **humanoid animation** system and **retargeting**, you need to have a **rigged** and **skinned** humanoid type mesh.

A character **Model** generally consists of polygons in 3D modeling software or converted to polygon or triangulated mesh, from a more complex mesh type before export.

In order to control the movement of a character, you must create a **joint hierarchy** or **skeleton** which defines the bones inside the Mesh and their movement in relation to one another. The process for creating such a **joint** hierarchy is known as **rigging**.

You must then connect the mesh or *skin* to the joint hierarchy. This defines which parts of the character mesh move when a given joint is animated. The process of connecting the skeleton to the mesh is known as **skinning**.



Stages for preparing a character (modeling, rigging, and skinning)

How to obtain humanoid models

There are three main ways to obtain humanoid models to use in the Unity Animation system:

Use a procedural character system or character generator such as *Poser*, *Makehuman* or *Mixamo* . Some of these systems can rig and skin your mesh (like Mixamo) while others cannot. Furthermore, if you use these methods, you may need to reduce the number of polygons in your original mesh to make it suitable for use in Unity.

Purchase demo examples and character content from the Unity Asset Store.

You can also prepare your own character from scratch by following three steps: modeling, rigging and skinning.

Modeling

This is the process of creating your own humanoid Mesh in 3D modeling software (such as Autodesk® 3ds Max®, Autodesk® Maya®, or Blender). Although this is a whole subject in its own right, there are a few guidelines you can follow to ensure a Model works well with animation in a Unity Project:

Use a topology with a **well-formed structure**. The exact nature of a "well-formed" structure for your Mesh is rather subtle but generally, you should bear in mind how the vertices and triangles of the model are distorted as it is animated. A poor topology does not allow the Model to move without distorting the Mesh. Study existing 3D character Meshes to see how the topology is arranged and why.

Check the **scale** of your mesh. Do a test import and compare the size of your imported Model with a "meter cube". The standard Unity cube primitive has a side length of one unit, so it can be taken as a 1m cube for most purposes. Check the units your 3D modeling software uses and adjust the export settings so that the size of the Model is in correct proportion to the cube. It is easy to create models without any notion of their scale and consequently end up with a set of objects that are disproportionate in size when you imported them into Unity.

Arrange the mesh so that the character's feet are standing on the local origin or "anchor point" of the model. Since a character typically walks upright on a floor, it is much easier to handle if its anchor point (that is, its transform position) is directly on that floor.

Model in a **T-pose** if possible. This gives you space to refine polygon detail where you need it (such as the underarms). This also makes it easier to position your rig inside the Mesh.

While you are building, **clean up your Model**. Where possible, cap holes, weld verts, and remove hidden faces. This helps with skinning, especially automated skinning processes.

Skin Mesh, textured and triangulated

Rigging

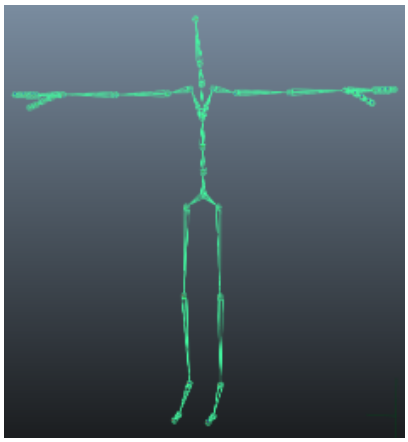
This is the process of creating a skeleton of joints to control the movements of your Model.

3D modeling software provides a number of ways to create joints for your humanoid rig. These range from ready-made biped skeletons that you can scale to fit your Mesh, right through to tools for individual bone creation and parenting to create your own bone structure. To work with animation in Unity, make sure they are the root element of the bone hierarchy. A minimum of fifteen bones are required in the skeleton.

Your skeleton needs to have at least the required bones in place for Unity to produce a valid match. In order to improve your chances for finding a match to the **Avatar**, name your bones in a way that reflects the body parts they represent. For example, “LeftArm” and “RightForearm” make it clear what these bones control.

The joint/bone hierachy should follow a natural structure for the character you are creating. Given that arms and legs come in pairs, you should use a consistent convention for naming them (for example, “arm_L” for the left arm, “arm_R” for the right arm). Possible structures for the hierarchy include:

```
* HIPS - spine - chest - shoulders - arm - forearm - hand
* HIPS - spine - chest - neck - head
* HIPS - UpLeg - Leg - foot - toe - toe_end
```



Biped Skeleton, positioned in T-pose

Skinning

This is the process of attaching the Mesh to the skeleton.

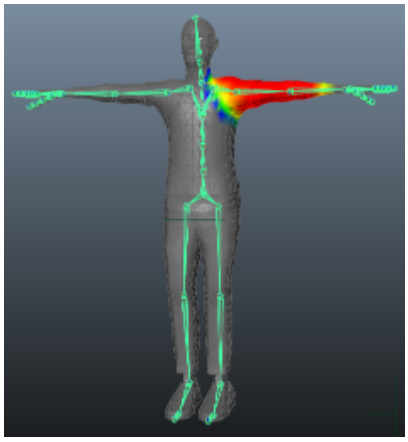
Skinning involves binding vertices in your Mesh to bones, either directly (rigid bind) or with blended influence to a number of bones (soft bind). Different 3D modeling software uses different methods. for example, you can assign individual vertices and paint the weighting of influence per bone onto the Mesh. The initial setup is typically automated, say by finding the nearest influence or using "**heatmaps**". Skinning usually requires a fair amount of work and testing with animations in order to ensure satisfactory results for the skin deformation. Some general guidelines for this process include:

Use an automated process initially to set up some of the skinning (see the skinning tutorials available for your 3D modeling software).

Create a simple animation for your rig or import some animation data to act as a test for the skinning. This should give you a quick way to evaluate whether or not the skinning looks good in motion.

Incrementally edit and refine your skinning solution.

Limit the number of influences when using a soft bind to a maximum of four, since this is the maximum number that Unity supports. If your Mesh uses more than four influences, then you lose some information when playing the animation in Unity.



Interactive Skin Bind, one of many skinning methods

Export and verify

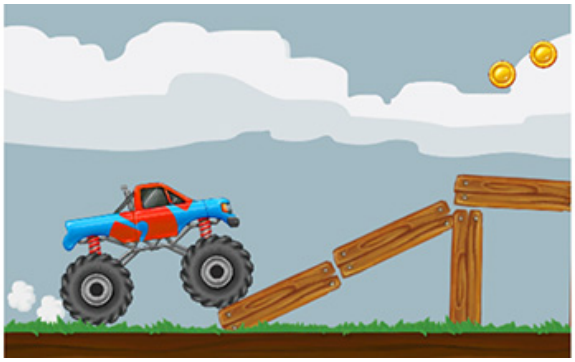
Unity imports a number of different generic and native 3D file formats. FBX is the recommended format for exporting and verifying your Model since you can use it to:

Export the Mesh with the skeleton hierarchy, normals, textures and animation.
Re-import the Mesh into your 3D modeling software to verify your animated Model looks as expected.
Export animations without the Mesh.

2018–04–25 Page amended with limited editorial review

2D

Leave feedback



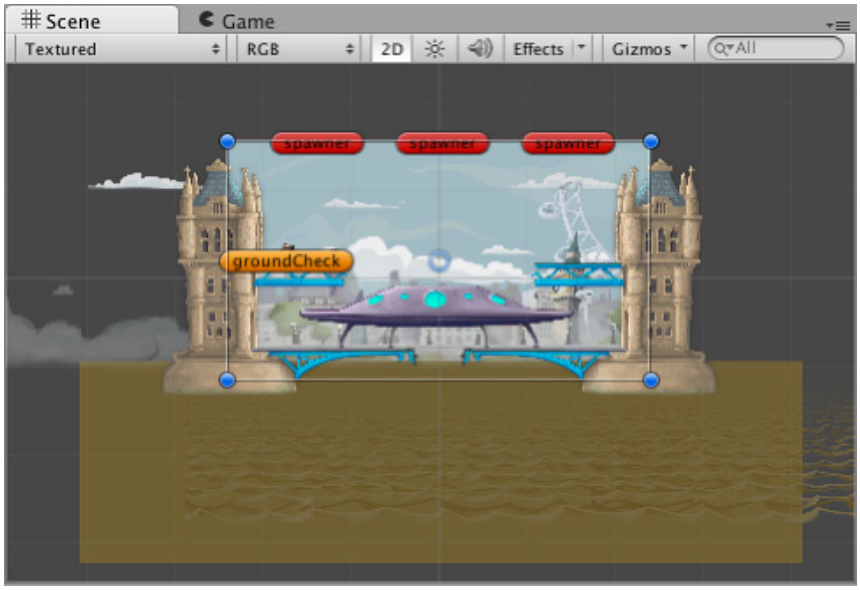
This section contains documentation for users developing 2D games in Unity. Note that many areas of the Unity documentation apply to both 2D and 3D development; this section focuses solely on 2D-specific features and functionality.

See documentation on 2D and 3D mode settings for information on how to change the 2D/3D mode how the modes differ. See 2D or 3D Projects if you're not sure whether you should be working in 2D or 3D.

Gameplay in 2D

Leave feedback

While famous for its 3D capabilities, Unity can also be used to create 2D games. The familiar functions of the editor are still available but with helpful additions to simplify 2D development.



Scene viewed in 2D mode

The most immediately noticeable feature is the *2D* view mode button in the **toolbar** of the **Scene view**. When 2D mode is enabled, an orthographic (ie, perspective-free) view will be set; the **camera** looks along the Z axis with the Y axis increasing upwards. This allows you to visualise the **scene** and place **2D objects** easily.

For a full list of 2D **components**, how to switch between 2D and 3D mode, and the different 2D and 3D Mode settings, see 2D or 3D Projects.

2D graphics

Graphic objects in 2D are known as **Sprites**. Sprites are essentially just standard textures but there are special techniques for combining and managing sprite textures for efficiency and convenience during development. Unity provides a built-in Sprite Editor to let you extract sprite graphics from a larger image. This allows you to edit a number of component images within a single texture in your image editor. You could use this, for example, to keep the arms, legs and body of a character as separate elements within one image.

Sprites are rendered with a Sprite Renderer component rather than the Mesh Renderer used with 3D objects. You can add this to a GameObject via the Components menu (**Component > Rendering > Sprite Renderer** or alternatively, you can just create a GameObject directly with a Sprite Renderer already attached (menu: **GameObject > 2D Object > Sprite**).

In addition, you can use a Sprite Creator tool to make placeholder 2D images.

2D physics

Unity has a separate **physics engine** for handling 2D physics so as to make use of optimizations only available with 2D. The components correspond to the standard 3D physics components such as Rigidbody, Box Collider and Hinge Joint, but with "2D" appended to the name. So, sprites can be equipped with Rigidbody 2D, Box Collider 2D and Hinge Joint 2D. Most 2D physics components are simply "flattened" versions of the 3D equivalents (eg, *Box Collider 2D* is a square while *Box Collider* is a cube) but there are a few exceptions.

For a full list of 2D physics components, see 2D or 3D Projects. See the Physics section of the manual for further information about 2D physics concepts and components. To specify 2D physics settings, see Physics 2D Settings.

2018–04–24 Page amended with limited editorial review

Sprites

Leave feedback

Sprites are 2D Graphic objects. If you are used to working in 3D, **Sprites** are essentially just standard textures but there are special techniques for combining and managing sprite textures for efficiency and convenience during development.

Unity provides a placeholder Sprite Creator, a built-in Sprite Editor, a Sprite Renderer and a Sprite Packer

See **Importing and Setting up Sprites** below for information on setting up assets as **Sprites** in your Unity project.

Sprite Tools

Sprite Creator

Use the Sprite Creator to create placeholder sprites in your project, so you can carry on with development without having to source or wait for graphics.

Sprite Editor

The Sprite Editor lets you extract sprite graphics from a larger image and edit a number of **component** images within a single texture in your image editor. You could use this, for example, to keep the arms, legs and body of a character as separate elements within one image.

Sprite Renderer

Sprites are rendered with a Sprite Renderer component rather than the Mesh Renderer used with **3D objects**. Use it to display images as **Sprites** for use in both 2D and 3D **scenes**.

Sprite Packer

Use Sprite Packer to opimize the use and performance of video memory by your project.

Importing and Setting Up Sprites

Sprites are a type of **Asset** in Unity projects. You can see them, ready to use, via the **Project** view.

There are two ways to bring **Sprites** into your project:

In your computer's Finder (Mac OS X) or File Explorer (Windows), place your image directly into your Unity Project's **Assets** folder.

Unity detects this and displays it in your project's **Project** view.

In Unity, go to **Assets** > **Import New Asset** to bring up your computer's Finder (Mac OS X) or File Explorer (Windows).

From there, select the image you want, and Unity puts it in the **Project** view.

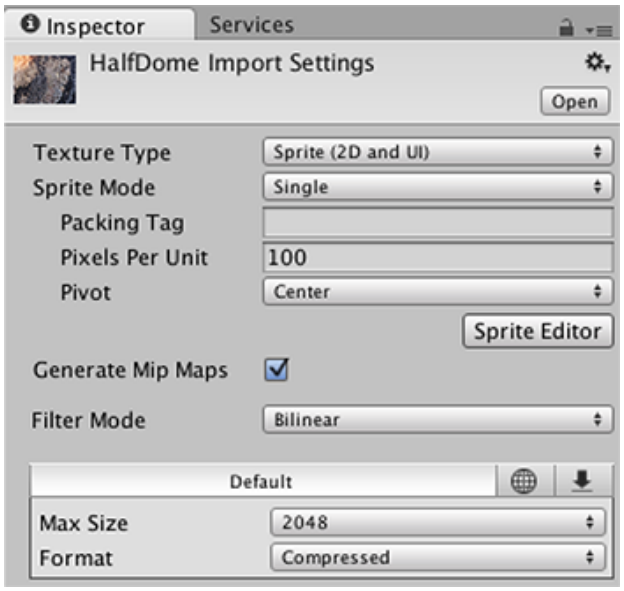
See Importing for more details on this and important information about organising your **Assets** folder.

Setting your Image as a Sprite

If your project mode is set to 2D, the image you import is automatically set as a **Sprite**. For details on setting your project mode to 2D, see 2D or 3D Projects.

However, if your project mode is set to 3D, your image is set as a **Texture**, so you need to change the asset's **Texture Type**:

Click on the asset to see its **Import Inspector**.
Set the **Texture Type** to **Sprite (2D and UI)**:



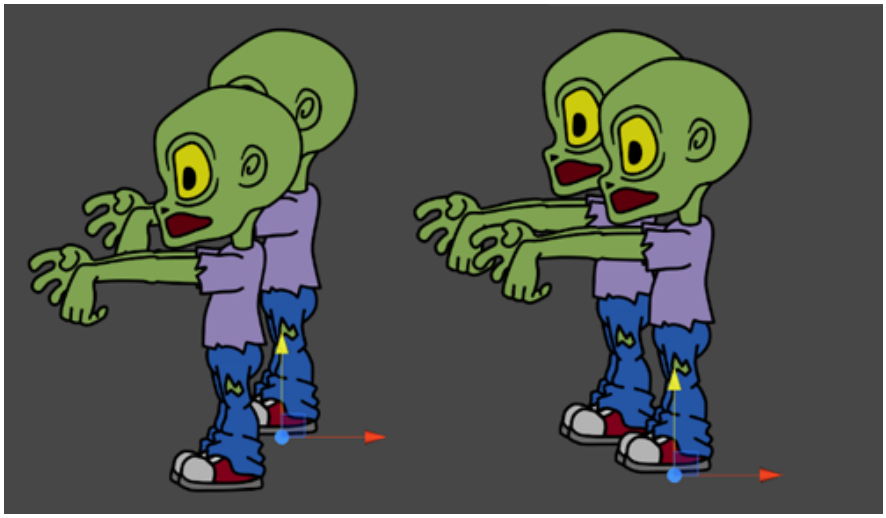
Set Texture Type to Sprite (2D and UI) in the Asset's Inspector

For details on Sprite **Texture Type** settings, see Texture type: Sprite (2D and UI).

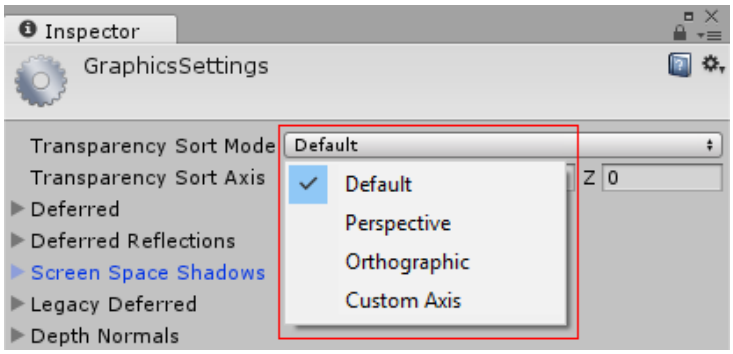
Sorting Sprites

Renderers in Unity are sorted by several criteria, such as their Layer order or their distance from the Camera. Unity's GraphicsSettings (menu: **Edit** > **Project Settings** > **Graphics**) provide a setting called **Transparency Sort Mode**, which allows you to control how Sprites are sorted depending on where they are in relation to the Camera. More specifically, it uses the Sprite's position on an axis to determine which ones are transparent against others, and which are not.

An example of when you might use this setting is to sort Sprites along the Y axis. This is quite common in 2D games, where Sprites that are higher up are sorted behind Sprites that are lower, to make them appear further away.



There are four **Transparency Sort Mode** options available:



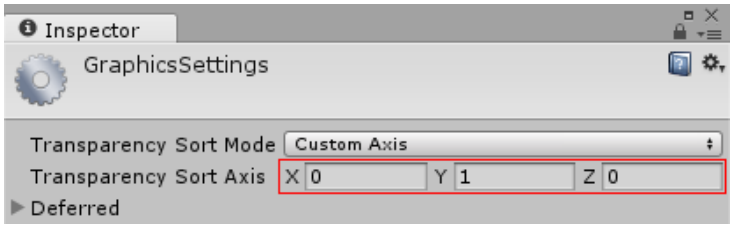
Default - Sorts based on whether the **Camera**'s **Projection** mode is set to **Perspective** or **Orthographic**

Perspective - Sorts based on perspective view. Perspective view sorts Sprites based on the distance from the Camera's position to the Sprite's center.

Orthographic - Sorts based on orthographic view. Orthographic view sorts Sprites based on the distance along the view direction.

Custom Axis - Sorts based on the given axis set in Transparency Sort Axis

If you have set the **Transparency Sort Mode** to **Custom**, you then need to set the **Transparency Sort Axis**:



If the **Transparency Sort Mode** is set to **Custom Axis**, renderers in the **Scene view** are sorted based on the distance of this axis from the camera. Use a value between –1 and 1 to define the axis. For example: X=0, Y=1, Z=0 sets the axis direction to up. X=1, Y=1, Z=0 sets the axis to a diagonal direction between X and Y.

For example, if you want Sprites to behave like the ones in the image above (those higher up the y axis standing behind the Sprites that are lower on the axis), set the **Transparency Sort Mode** to **Custom Axis**, and set the **Y** value for the **Transparency Sort Axis** to a value higher than 0.

Sorting Sprites using script

You can also sort Sprites per camera through **scripts**, by modifying the following properties in Camera:

TransparencySortMode (corresponds with **Transparency Sort Mode**)

TransparencySortAxis (corresponds with **Transparency Sort Axis**)

For example:

```
var camera = GetComponent<Camera>();

camera.transparencySortMode = TransparencySortMode.CustomAxis;

camera.transparencySortAxis = new Vector3(0.0f, 1.0f, 0.0f);
```

2018–04–25 Page amended with limited editorial review

2017–05–24 Page amended with no editorial review

Transparancy Sort Mode added in 5.6

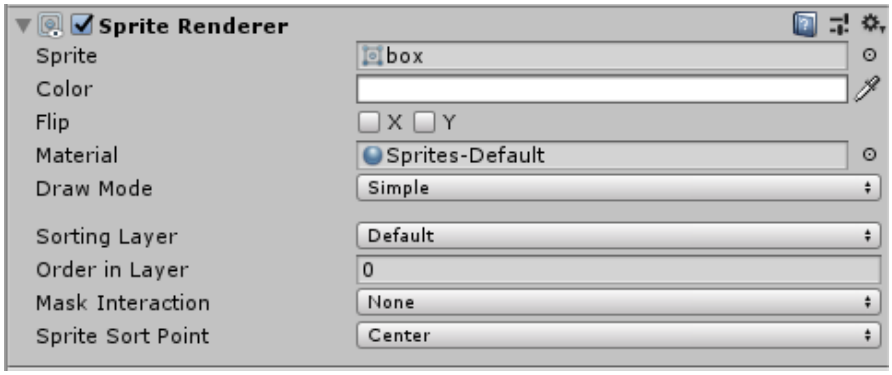
Sprite Renderer

Leave feedback

The **Sprite Renderer** component renders the **Sprite** and controls how it visually appears in a **Scene** for both 2D and 3D projects.

When you create a Sprite (**GameObject > 2D Object > Sprite**), Unity automatically creates a GameObject with the **Sprite Renderer** component attached. You can also add the component to an existing GameObject via the **Components** menu (**Component > Rendering > Sprite Renderer**).

Properties



Sprite Renderer Inspector

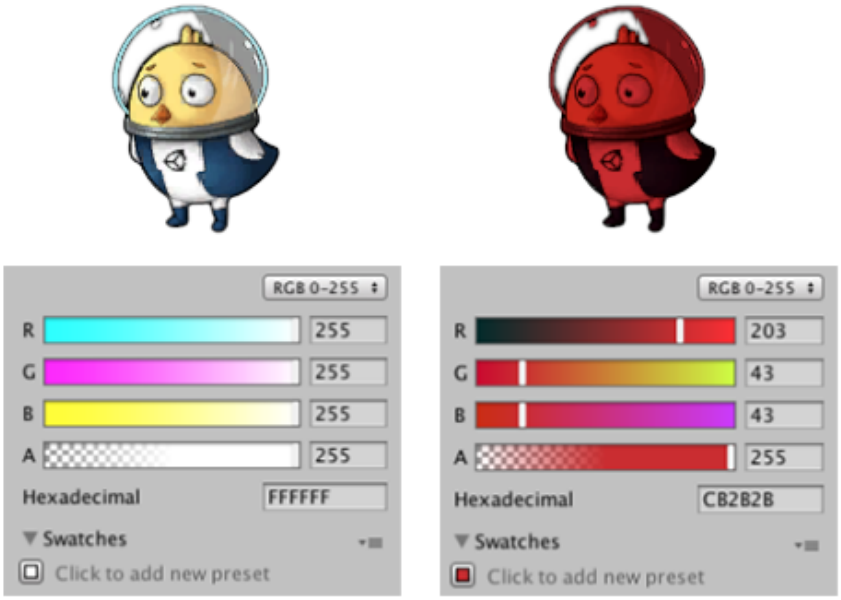
Property	*Function*
Sprite	Define which Sprite texture the component should render. Click the small dot to the right to open the object picker window, and select from the list of available Sprite **Assets**.
Color	Define the vertex color of the Sprite, which tints or recolors the Sprite's image. Use the color picker to set the vertex color of the rendered Sprite texture. See the Color section below this table for examples.
Flip	Flips the Sprite texture along the checked axis. This does not flip the Transform position of the **GameObject**.
Material	Define the Material used to render the Sprite texture.
Draw Mode	Define how the Sprite scales when its dimensions change. Select one of the following options from the drop-down box.
Simple	The entire image scales when its dimensions change. This is the default option.
Sliced	Select this mode if the Sprite is 9-sliced.
Size *('Sliced' or 'Tiled')*	Enter the Sprite's new Width and Height to scale the 9-sliced Sprite correctly. You can also use the Rect Transform Tool to scale the Sprite while applying 9-slicing properties.
Tiled	By default, this mode causes the middle of the 9-Sliced Sprite to tile instead of scale when its dimensions change. Use **Tile Mode** to control the tiling behavior of the Sprite.
Continuous	This is the default **Tile Mode**. In Continuous mode, the midsection tiles evenly when the Sprite dimensions change.
Adaptive	In Adaptive mode, the Sprite texture stretches when its dimensions change, similar to Simple mode. When the scale of the changed dimensions meets the Stretch Value, the midsection begins to tile.
Stretch Value	Use the slider to set the value between 0 and 1. The maximum value is 1, which represents double the original Sprite's scale.
Sorting Layer	Set the Sorting Layer of the Sprite, which controls its priority during **rendering**. Select an existing Sorting Layer from the drop-down box, or create a new Sorting Layer.

Property	*Function*
Order In Layer	Set the render priority of the Sprite within its Sorting Layer. Lower numbered Sprites are rendered first, with higher numbered Sprites overlapping those below.
Mask Interaction	Set how the Sprite Renderer behaves when interacting with a Sprite Mask. See examples of the different options in the Mask Interaction section below.
None	The Sprite Renderer does not interact with any Sprite Mask in the Scene. This is the default option.
Visible Inside Mask	The Sprite is visible where the Sprite Mask overlays it, but not outside of it.
Visible Outside Mask	The Sprite is visible outside the Sprite Mask, but not inside it. The Sprite Mask hides the sections of the Sprite it overlays.
Sprite Sort Point	Choose between the Sprite's Center or its Pivot Point when calculating the distance between the Sprite and the **camera**. See the section on Sprite Sort Point for further details.

Details

Color

The image below demonstrates the effect of changing the RGB values on the **Sprite Renderer's** Color setting. To change a Sprite's opacity, change the value of its Color property's Alpha (A) channel.



Left: The original Sprite. **Right**: The Sprite with its RGB colors set to red.

Material

Use a Material's Material and Shader settings to control how Unity renders it. Refer to Materials, Shaders & Textures for further information on these settings.

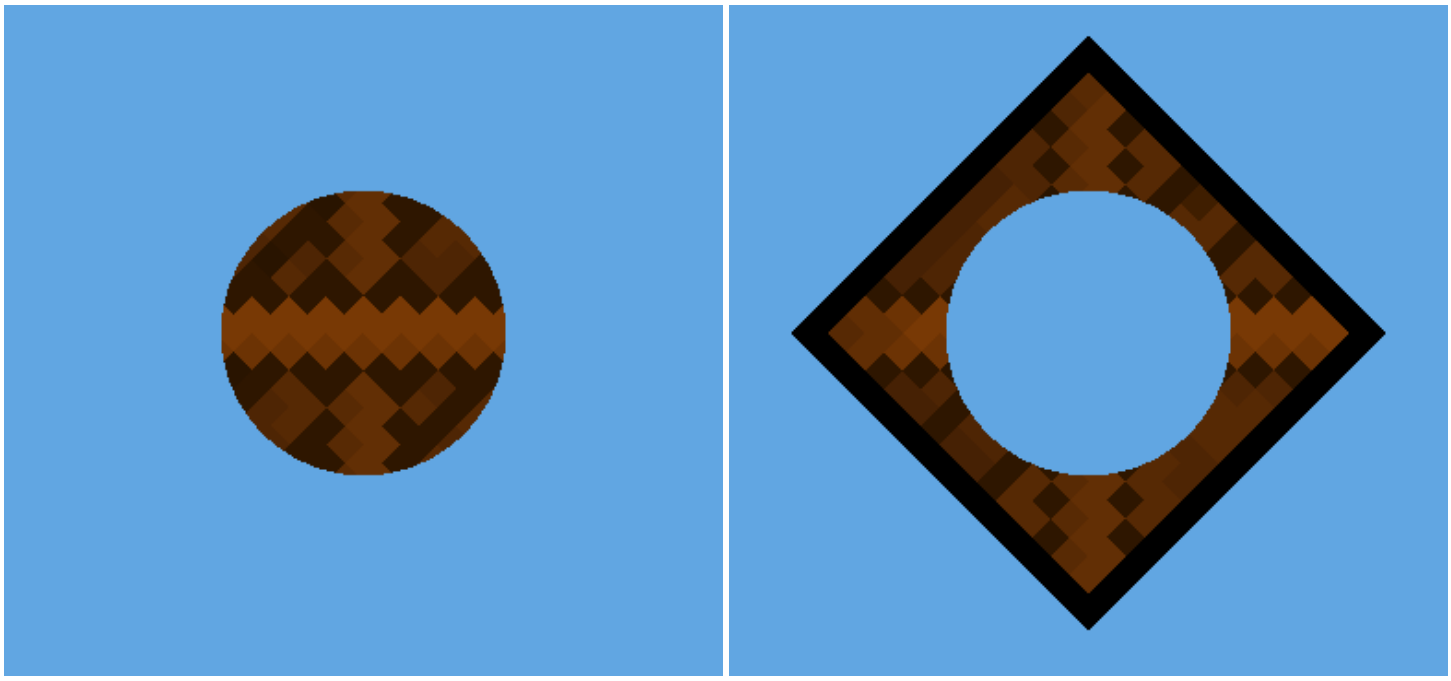
The default Material for newly created Sprites is **Sprites - Default**. Scene lighting does not affect this default Sprite. To have the Sprite react to lighting, assign the Material **Default - Diffuse** instead. To do this, click the small circle next to the Material field to bring up the object picker window, and select the **Default-Diffuse** Material.

Mask Interaction

Mask Interaction controls how the Sprite Renderer interacts with Sprite Masks. Select either **Visible Inside Mask** or **Visible Outside Mask** from the drop-down menu. The examples below demonstrate the effect of each option with a square Sprite

and a circle Mask:

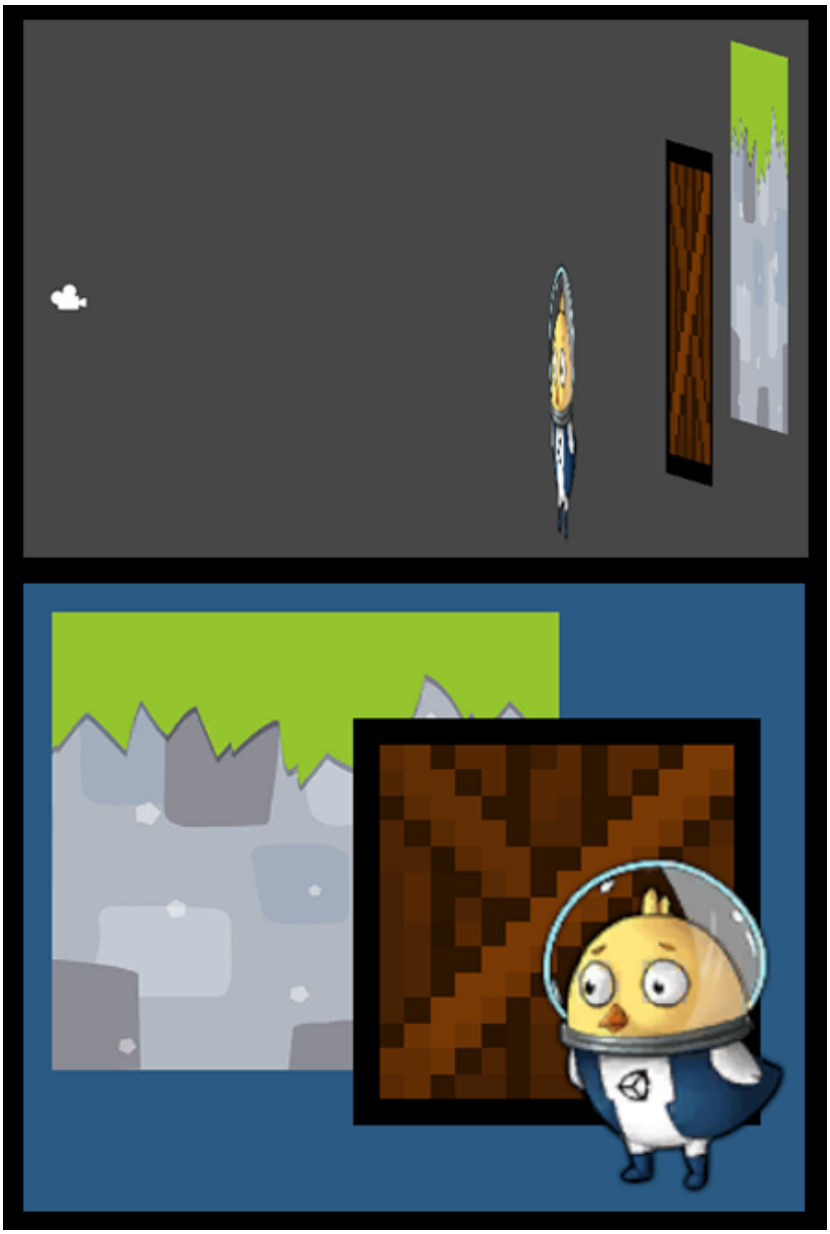
To interact with a Sprite Mask, select **Visible Inside Mask** or **Visible Outside Mask** from the drop-down menu.



Sprite Sort Point

This property is only available when the Sprite Renderer's **Draw Mode** is set to **Simple**.

In a 2D project, the Main Camera is set to Orthographic Projection mode by default. In this mode, Unity renders Sprites in the order of their their distance to the camera, along the direction of the Camera's view.



Orthographic Camera: Side view (top) and Game view (bottom)

By default, a Sprite's **Sort Point** is set to its *Center*, and Unity measures the distance between the camera's Transform position and the *Center* of the Sprite to determine their render order.

To set to a different **Sort Point** from the *Center*, select the **Pivot** option. Edit the Sprite's *Pivot* position in the Sprite Editor.

2018–10–05 Page amended with editorial review

Ability to sort Sprite-based renderers using the pivot position added in 2017.3

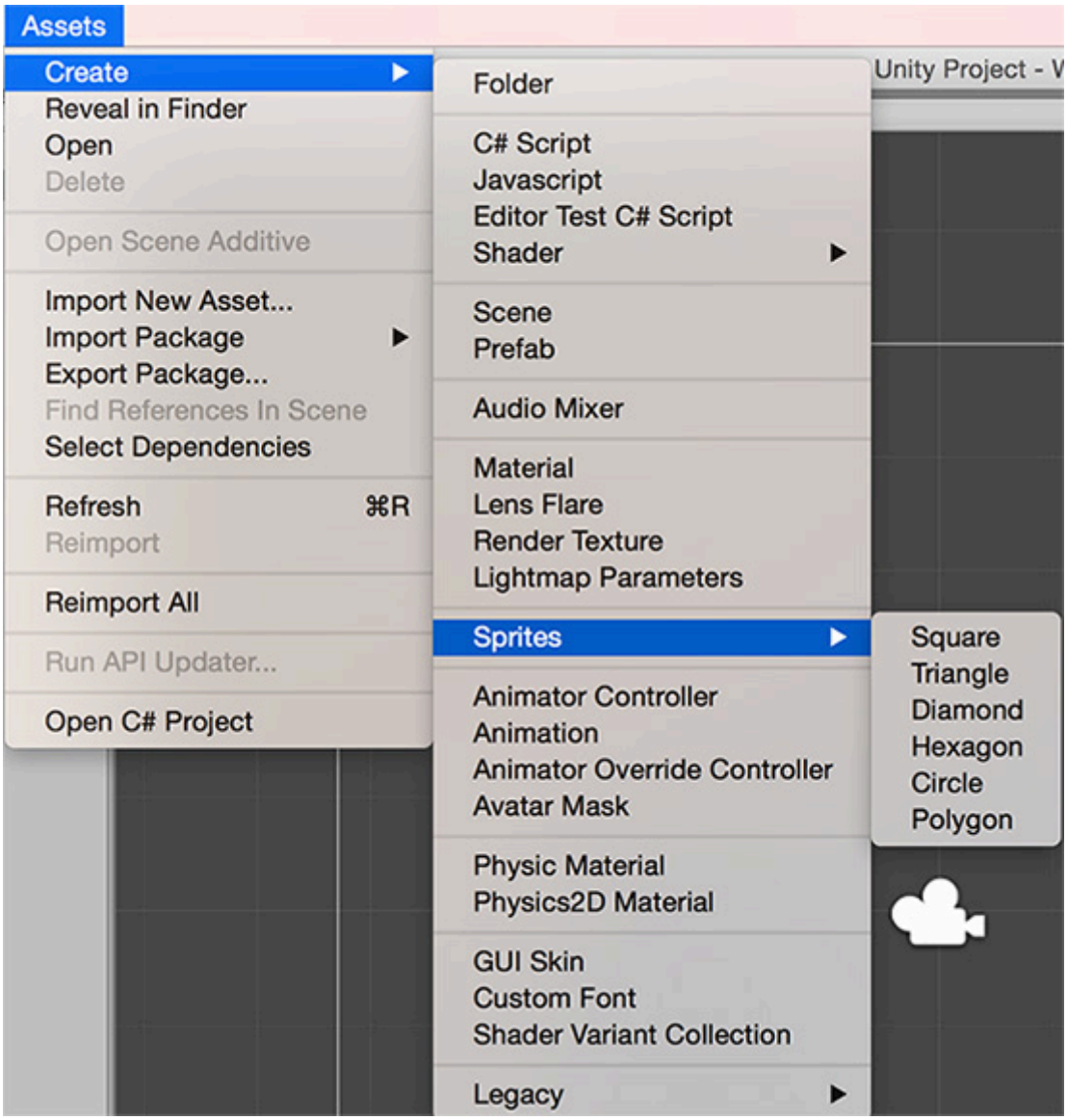
Sprite Creator

Leave feedback

With this tool you can create temporary placeholder **sprite** (2D) graphics. You can use these in your project during development and then replace them with the graphics you want to use.

Accessing the Sprite Creator

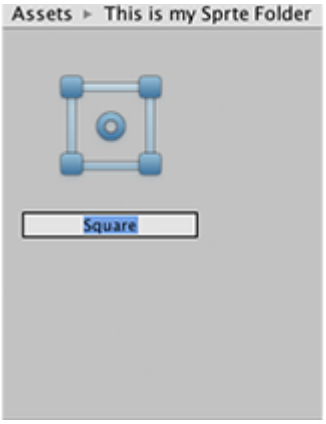
Select **Assets>Create>Sprites** and then select the placeholder sprite you want to make (square, triangle, diamond, hexagon, or polygon).

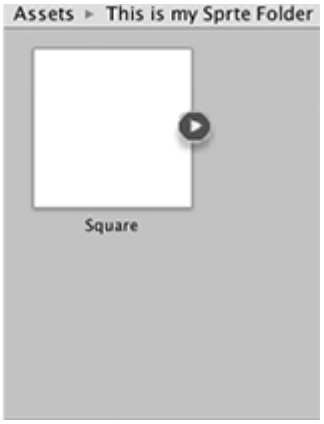


Accessing the Sprite Creator

Using the Sprite

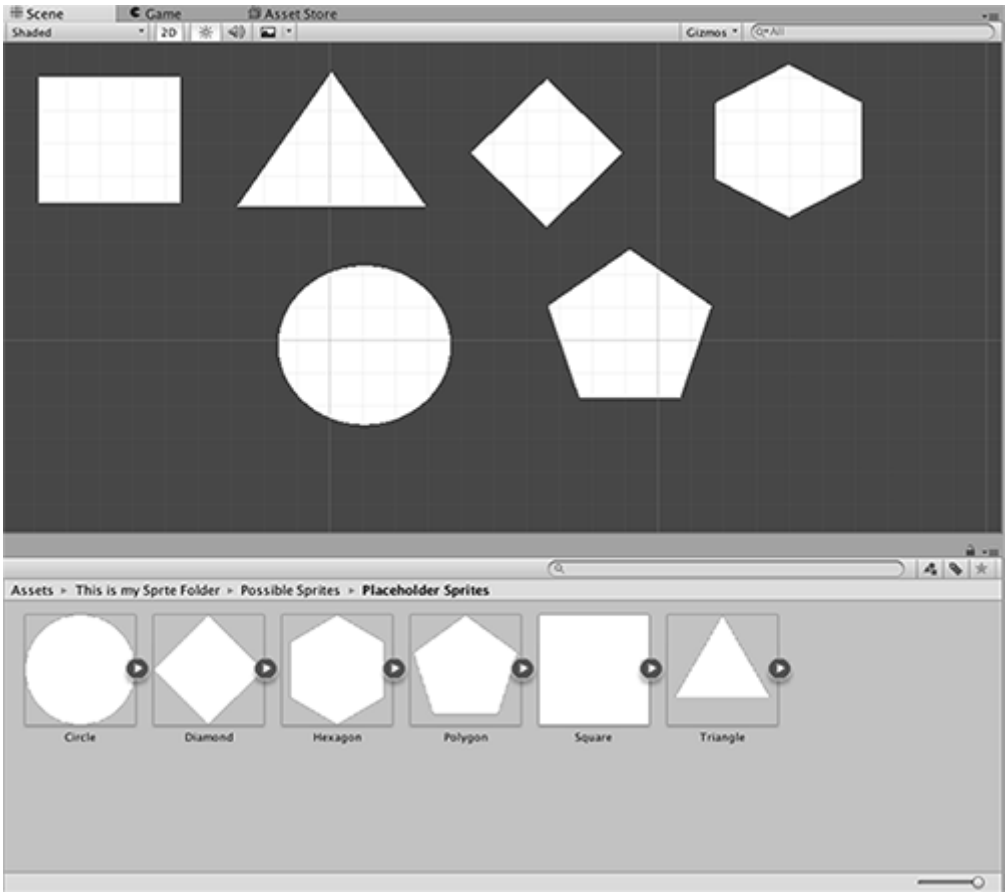
Your new placeholder sprite appears as a white shape in the **asset** folder you currently have open. The new sprite's name defaults to its shape name but you have the option to rename your sprite when it is first created. If you are not sure what you want to call it, leave it as the default; you can change it later by clicking on it.





Name your new sprite (or leave it to default)

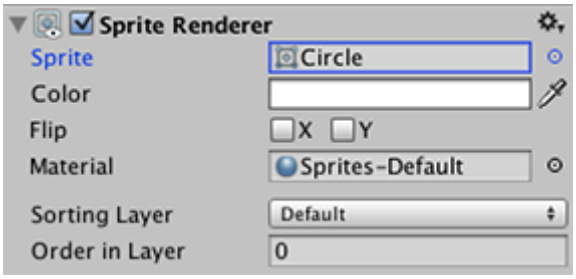
You can drag and drop your placeholder sprite into the **Scene View** or **Hierarchy** to start using it in your project.



Drag and drop your placeholder sprite into Scene View

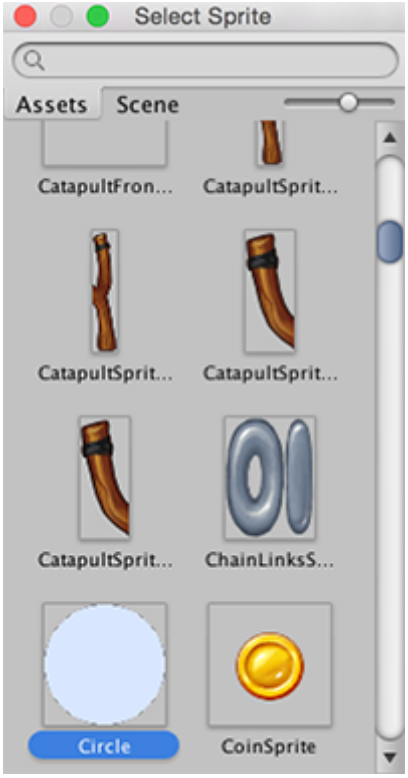
Replacing your Placeholder Sprite

To change your placeholder sprite, click on it in the **Scene View** and then edit via the **Sprite Renderer Component** in the **Inspector**.



Replace your sprite via the Sprite Renderer Component in the Inspector tool

Edit the **Sprite** field: You can click on the small circle to the right of the input field to bring up the **Sprite Selector** where you can browse and select from a menu of available 2D graphic assets.



Sprite Selector

Details

The Sprite Creator makes 4x4 white PNG outline textures.

The placeholder sprites are perfect primitive polygons (e.g. triangle, hexagon, n-sided polygon), generated by algorithm.

NOTE: Placeholder sprites are not like 3D primitives: A sprite is an asset, and as a many-sided polygon, may represent many different shapes, so placeholder sprites are not built like 3D primitives.

Sprite Editor

Leave feedback

Sometimes a sprite texture contains just a single graphic element but it is often more convenient to combine several related graphics together into a single image. For example, the image could contain **component** parts of a single character, as with a car whose wheels move independently of the body. Unity makes it easy to extract elements from a composite image by providing a **Sprite Editor** for the purpose.

NOTE:

Make sure the graphic you want to edit has its **Texture Type** set to **Sprite (2D and UI)**. For information on importing and setting up sprites, see Sprites.

Sprite textures with multiple elements need the **Sprite Mode** to be set to **Multiple** in the Inpsector. (See *Fig 2: Texture Import Inspector...* below.)

Opening the Sprite Editor

To open the **Sprite Editor**:

Select the 2D image you want to edit from the **Project View** *(Fig 1: Project View)*.

Note that you can't edit a sprite which is in the **Scene View**.

Click on the **Sprite Editor** button in the **Texture Import Inspector** *(Fig 2: Texture Import Inspector)* and the **Sprite Editor** displays *(Fig 3: Sprite Editor)*.

Note: You can only see the **Sprite Editor** button if the **Texture Type** on the image you have selected is set to **Sprite (2D and UI)**.

Fig 1: Project View

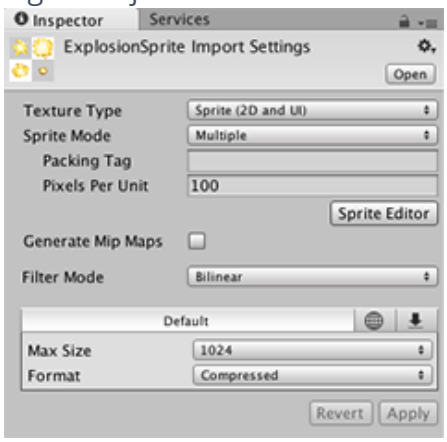


Fig 2: Texture Import Inspector with Sprite Editor button

Note: Set the **Sprite Mode** to **Multiple** in the **Texture Import Inspector** if your image has several elements.

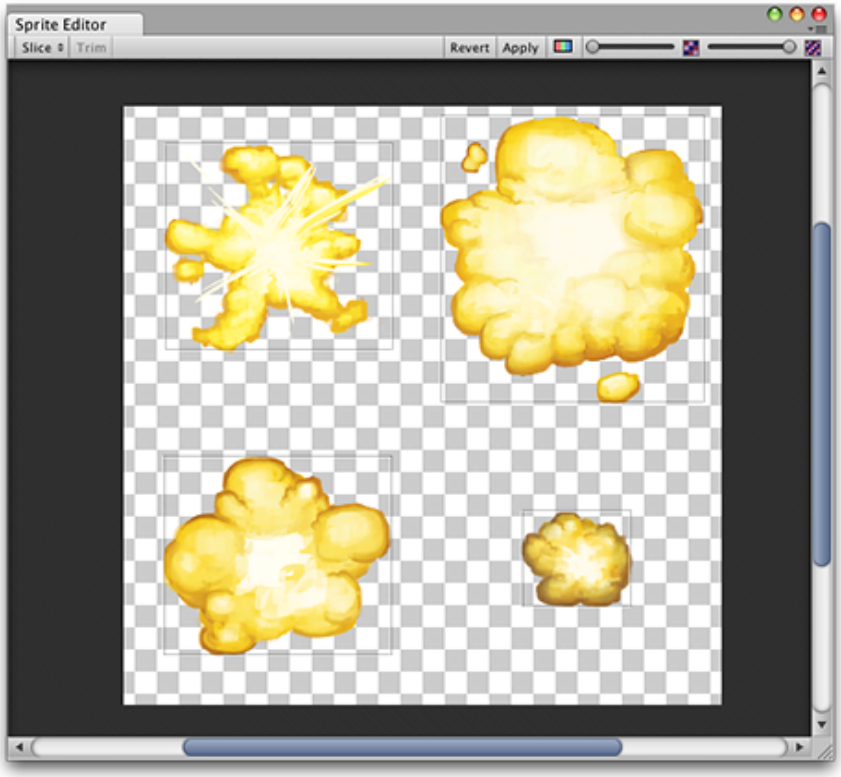
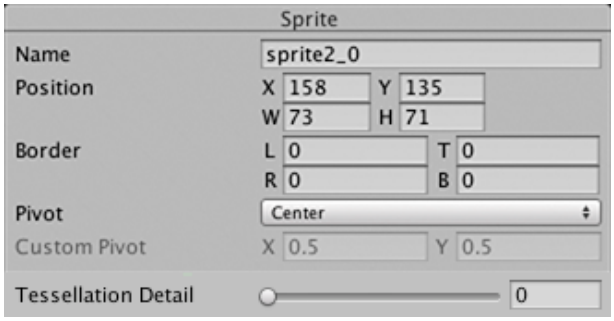


Fig 3: Sprite Editor

Along with the composite image, you will see a number of controls in the bar at the top of the window. The slider at the top right controls the zoom, while the color bar button to its left chooses whether you view the image itself or its alpha levels. The right-most slider controls the pixelation (mipmap) of the texure. Moving the slider to the left reduces the resolution of the sprite texture. The most important control is the **Slice** menu at the top left, which gives you options for separating the elements of the image automatically. Finally, the **Apply** and **Revert** buttons allow you to keep or discard any changes you have made.

Using the Editor

The most direct way to use the editor is to identify the elements manually. If you click on the image, you will see a rectangular selection area appear with handles in the corners. You can drag the handles or the edges of the rectangle to resize it around a specific element. Having isolated an element, you can add another by dragging a new rectangle in a separate part of the image. You'll notice that when you have a rectangle selected, a panel appears in the bottom right of the window:



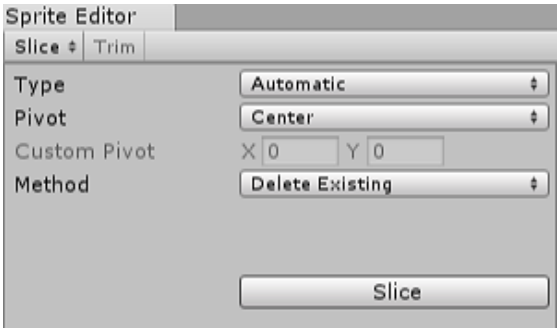
The controls in the panel let you choose a name for the sprite graphic and set the position and size of the rectangle by its coordinates. A border width, for left, top, right and bottom can be specified in **pixels**. There are also settings for the sprite's pivot, which Unity uses as the coordinate origin and main "anchor point" of the graphic. You can choose from a number of default rectangle-relative positions (eg, Center, Top Right, etc) or use custom coordinates.

The **Trim** button next to the Slice menu item will resize the rectangle so that it fits tightly around the edge of the graphic based on transparency.

Note: Borders are only supported for the **UI** system, not for the 2D SpriteRenderer.

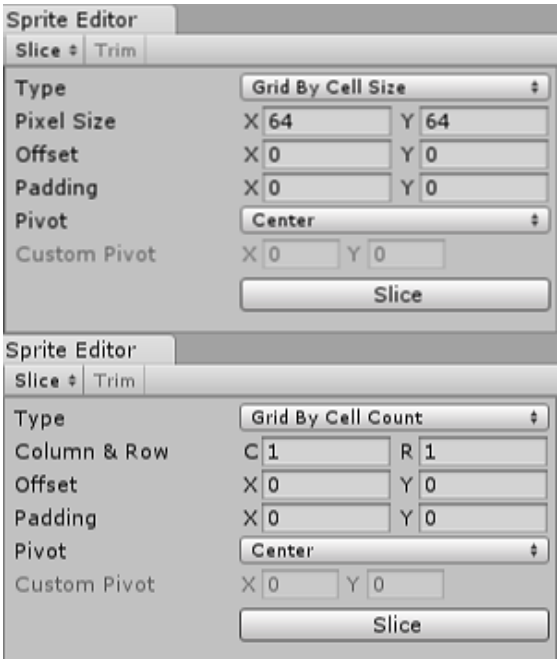
Automatic Slicing

Isolating the sprite rectangles manually works well but in many cases, Unity can save you work by detecting the graphic elements and extracting them for you automatically. If you click on the **Slice** menu in the control bar, you will see this panel:



With the slicing type set to **Automatic**, the editor will attempt to guess the boundaries of sprite elements by transparency. You can set a default pivot for each identified sprite. The **Method** menu lets you choose how to deal with existing selections in the window. The **Delete existing** option will simply replace whatever is already selected, **Smart** will attempt to create new rectangles while retaining or adjusting existing ones, and **Safe** will add new rectangles without changing anything already in place.

Grid by Cell Size or **Grid by Cell Count** options are also available for the slicing type. This is very useful when the sprites have already been laid out in a regular pattern during creation:



The **Pixel Size** values determine the height and width of the tiles in pixels. If you chose grid by cell count, **Column & Row** determines the number of columns and rows used for slicing. You can also use the **Offset** values to shift the grid position from the top-left of the image and the **Padding** values to inset the sprite rectangles slightly from the grid. The **Pivot** can be set with one of nine preset locations or a **Custom Pivot** location can be set.

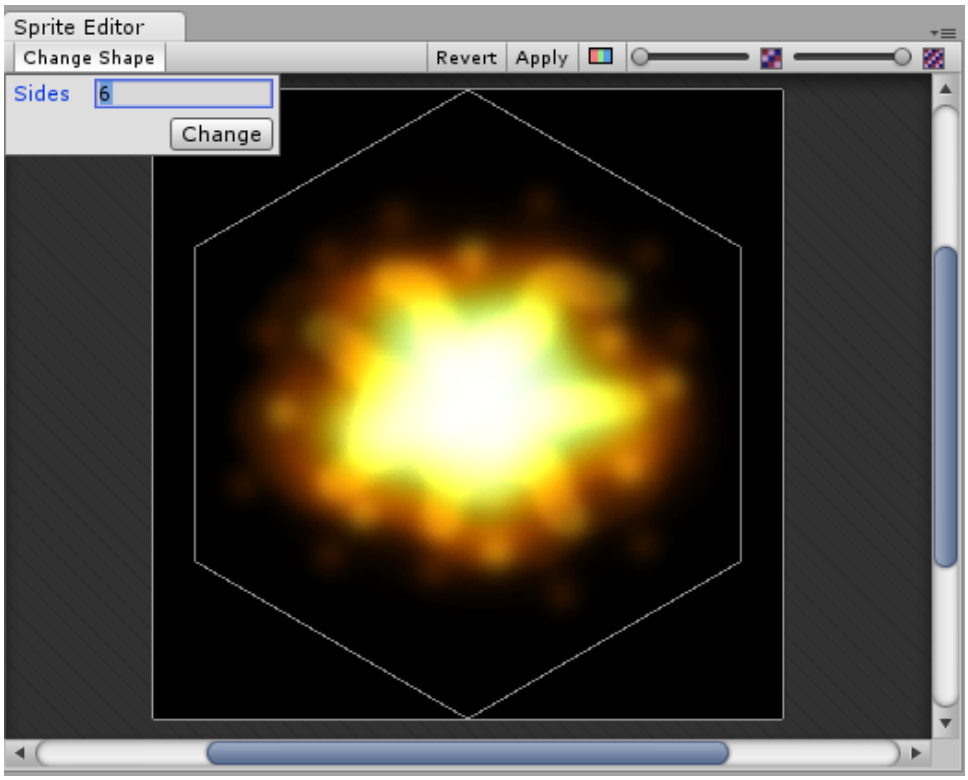
Note that after any of the automatic slicing methods has been used, the generated rectangles can still be edited manually. You can let Unity handle the rough definition of the sprite boundaries and pivots and then do any necessary fine tuning

yourself.

Polygon Resizing

Open the **Sprite Editor** for a polygon and you have the option to change its shape, size, and pivot position.

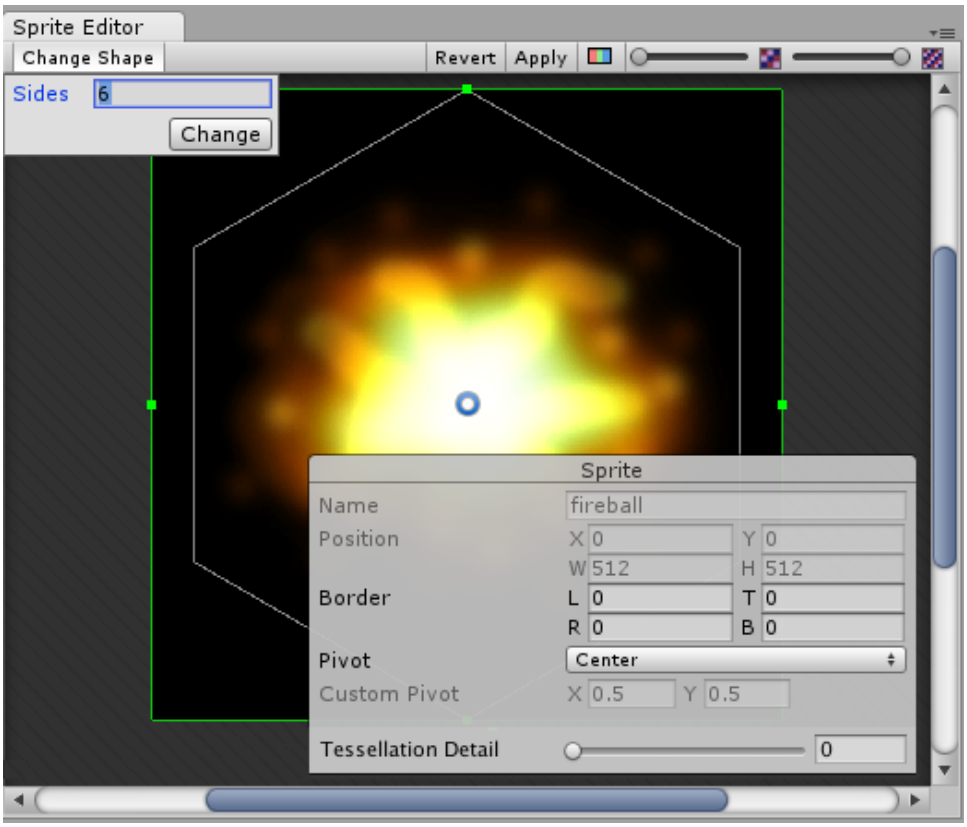
Shape



Sprite Editor: Polygon resizing - shape

Enter the number of sides you want the polygon to have in the **Sides** field and click **Change**.

Size and Pivot



Sprite Editor: Polygon resizing - size and pivot point - click on the polygon to display these options

SIZE: To change the polygon's size, click on the sprite to display green border lines and the Sprite information box. Click and drag on the green lines to create the border you want, and the values in the **Border** fields change. (Note that you cannot edit the **Border** fields directly.)

PIVOT: To change the polygon's pivot point (that is the axis point the polygon moves around), click on the image to display the Sprite information box. Click on the **Pivot** drop down menu and select an option. This displays a blue pivot circle on the polygon; its location depends on the pivot option to you have selected. If you want to change it further, select **Custom Pivot** and click and drag on the blue pivot circle to position it. (Note that you cannot edit the **Pivot** fields directly.)

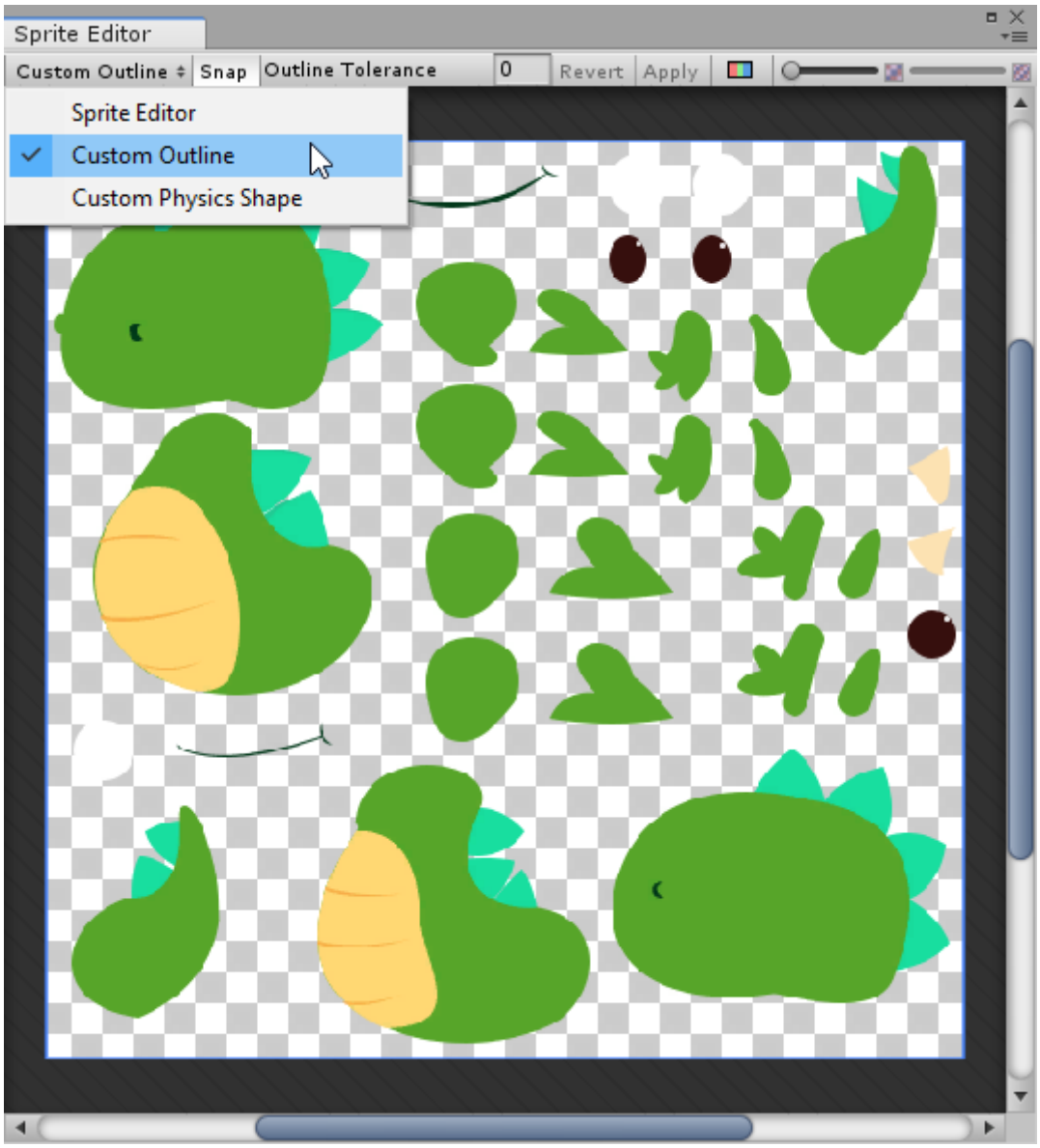
Sprite Editor: Edit Outline

Leave feedback

Use the Sprite Editor's **Edit Outline** option to edit the generated Mesh for a Sprite, effectively editing its outline.

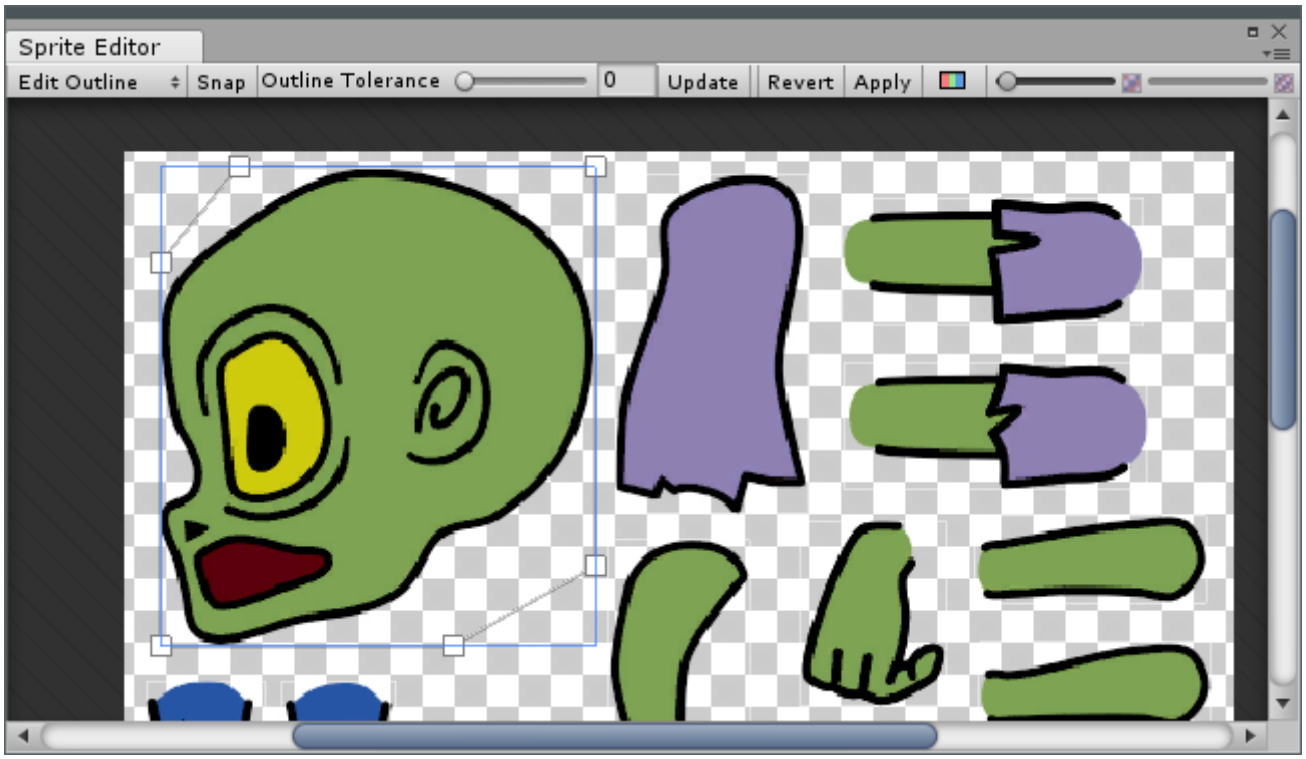
Transparent areas in a Sprite can negatively affect your project's performance. This feature is useful for fine-tuning the bounds of a Sprite, ensuring there are fewer transparent areas in the shape.

To access this option, select the Sprite and open the Sprite Editor (click **Sprite Editor** in the Inspector window). Click the **Sprite Editor** drop-down in the top left, and select **Edit Outline**.



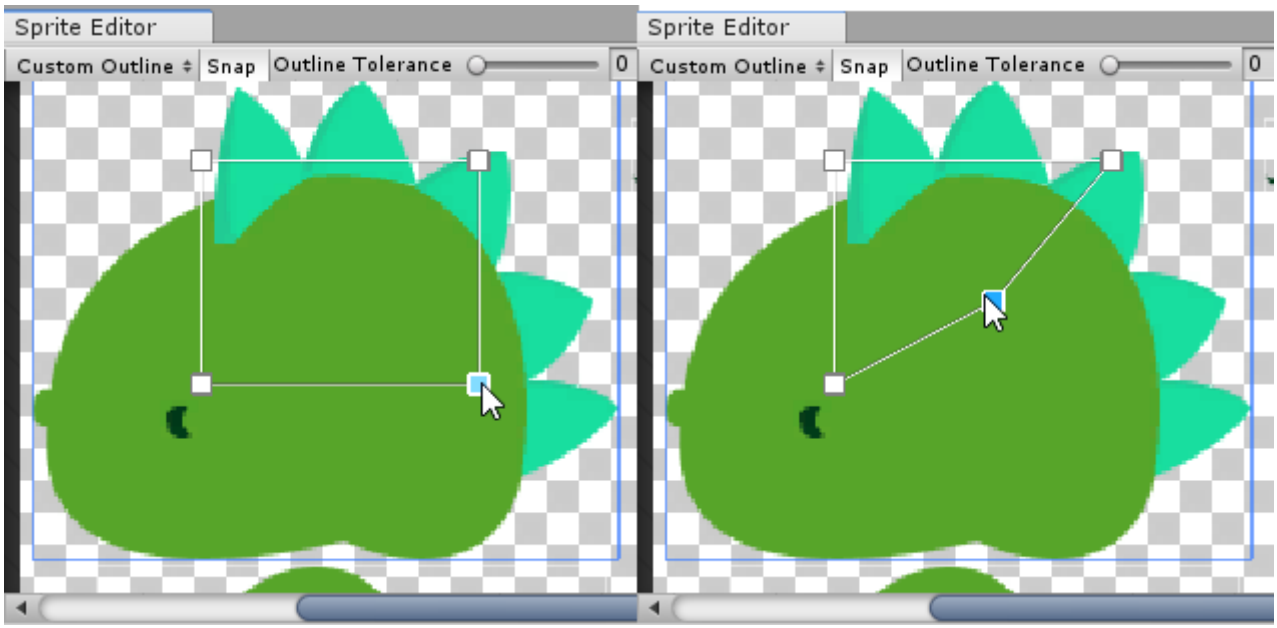
Use the Sprite Editor's drop-down to access **Edit Outline**

With **Edit Outline** selected, click a Sprite. The Sprite Editor displays the outline and control points of the Sprite. The outline is indicated by a white line. The control points are areas you can use to move and manipulate the outline. Control points are indicated by small squares. Click and drag a white square outline control point to change its position.



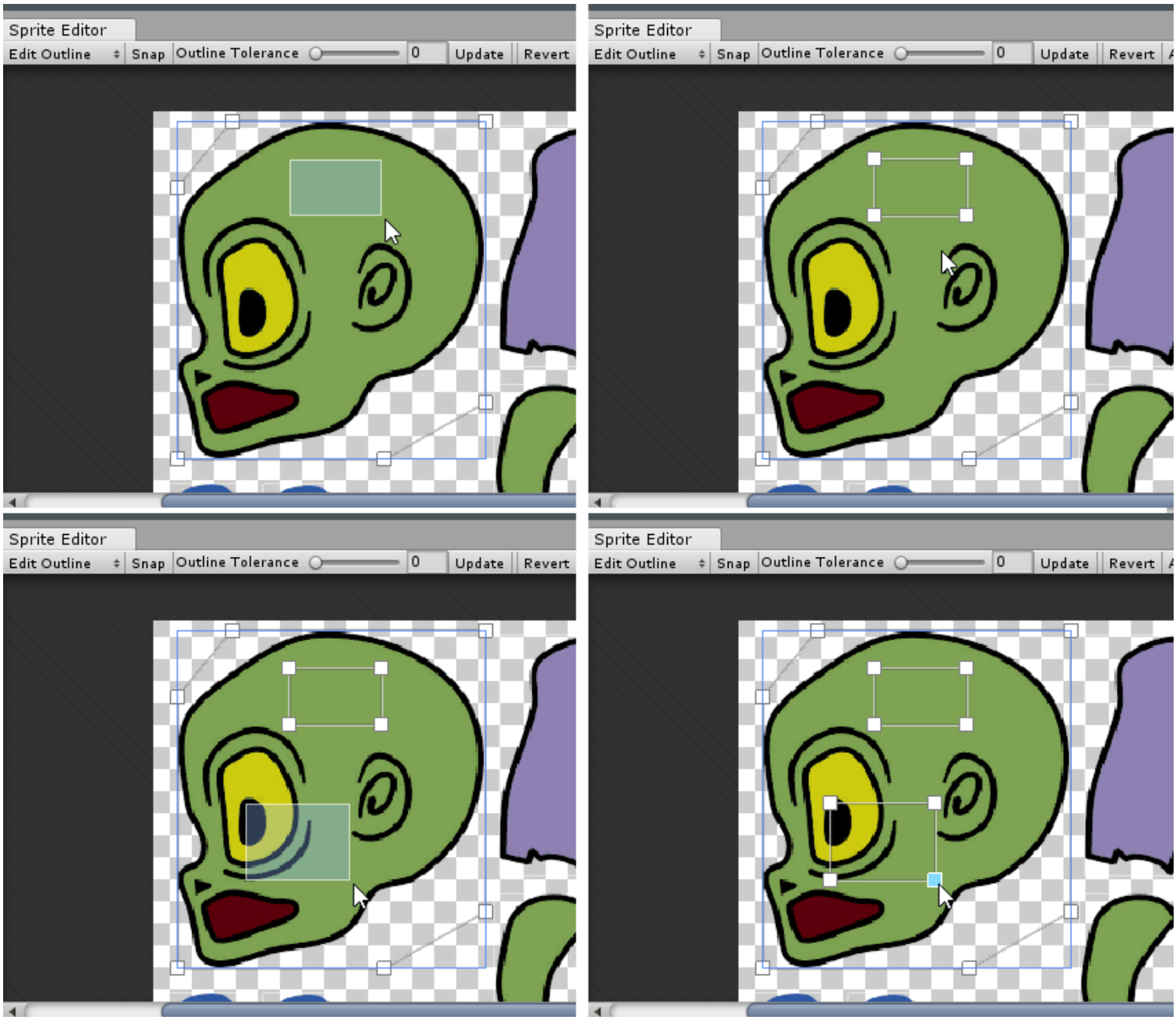
The white squares represent the outline control points - click and drag to change their positions

When you hover the mouse over a white square outline control point, a blue square appears on the outline. Drag this to reposition the control point and the blue square becomes a new white square outline control point, as shown below



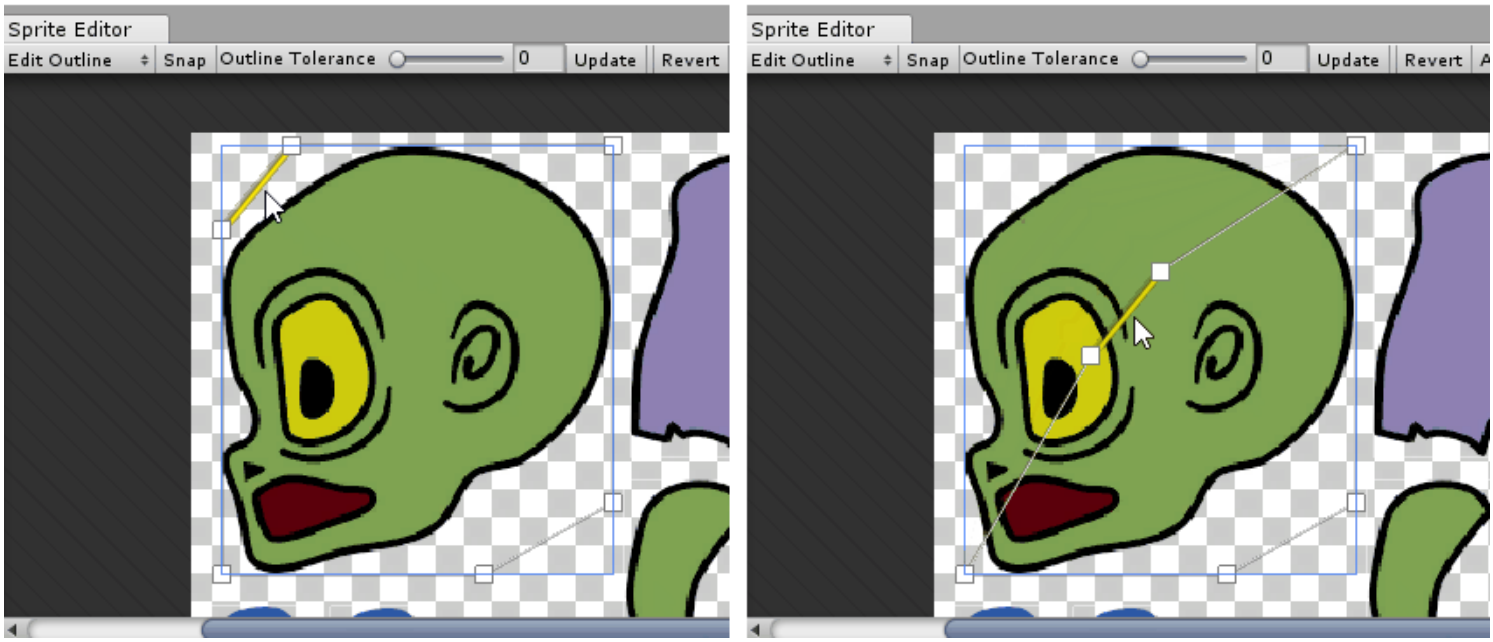
Hover over a white square control point, then drag the blue square to the position you want the new control point

To create new outlines, drag in an empty space within the Sprite. This creates a new rectangular outline with four control points. Do this multiple times to create multiple outlines on one Sprite (for example, a donut Sprite which has gaps inside the outline).



Create new outlines by dragging in an empty space within the Sprite - you can use multiple outlines in one Sprite

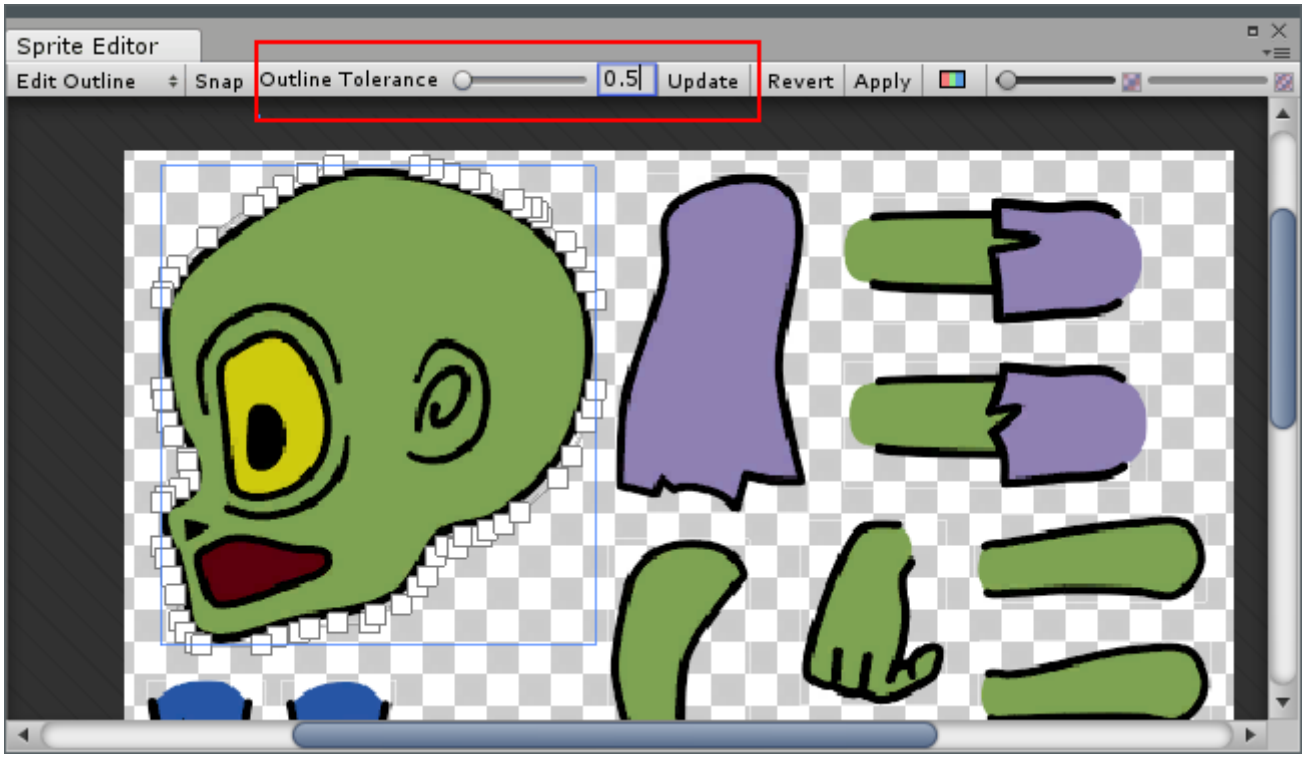
To move the outline instead of the control points, hold **Ctrl** while you click and drag the outline.



Hold **Ctrl** to drag the line (here shown in yellow) rather than the control points.

Outline Tolerance

Use the **Outline Tolerance** slider to increase and decrease the number of outline control points available, between **0** (the minimum number of control points) and **1** (the maximum number of control points). A higher **Outline Tolerance** creates more outline control points, while a lower **Outline Tolerance** creates a tighter Mesh (that is, a Mesh with a smaller border of transparent pixels between the Sprite and the Mesh edges). Click **Update** to apply the change.



Use the **Outline Tolerance** slider to determine the number of control points

The Sprite on the left has a low **Outline Tolerance**, and the Sprite on the right has a high **Outline Tolerance**

Sprite Editor: Custom Physics Shape

Leave feedback

Overview

The **Sprite** Editor's **Custom Physics Shape** allows you to edit a **Sprite**'s Physics Shape, which defines the initial shape of the **Sprite**'s Collider 2D **Mesh**. You can further refine the Physics Shape through the **Collider**'s **component** settings.

To edit a Sprite's Physics Shape:

In the Project window, select the Sprite that you want to change.

In the **Inspector** window, click the **Sprite Editor** button.

In the Sprite Editor window, select the top left drop-down menu and choose **Custom Physics Shape**.

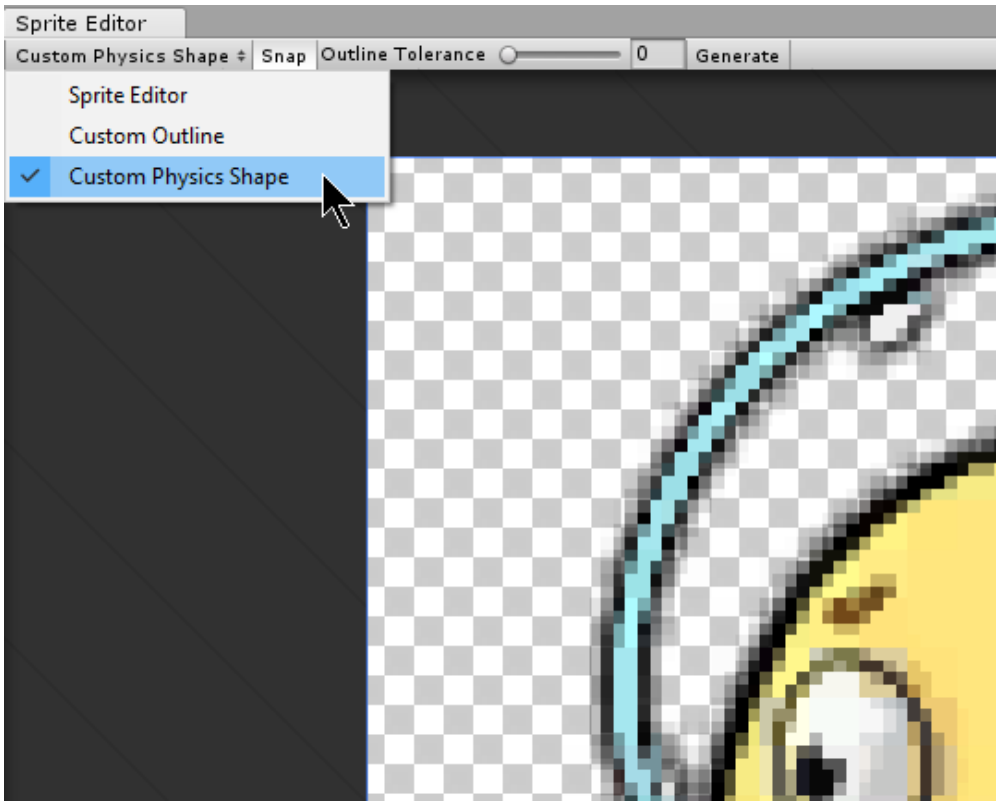
Editing a **Custom Physics Shape**

Properties

Property	Function
Snap	Snap control points to the nearest **pixel**.
Outline Tolerance	Use this to control how tightly the generated outline follows the outline of the Sprite texture. At the minimum value (0), the Sprite Editor generates a basic outline around the Sprite. At the maximum value (1), the Sprite Editor generates an outline that follows the pixel outline of the Sprite as closely as it can.
Generate	Click to automatically create a physics shape outline.

Standard workflow

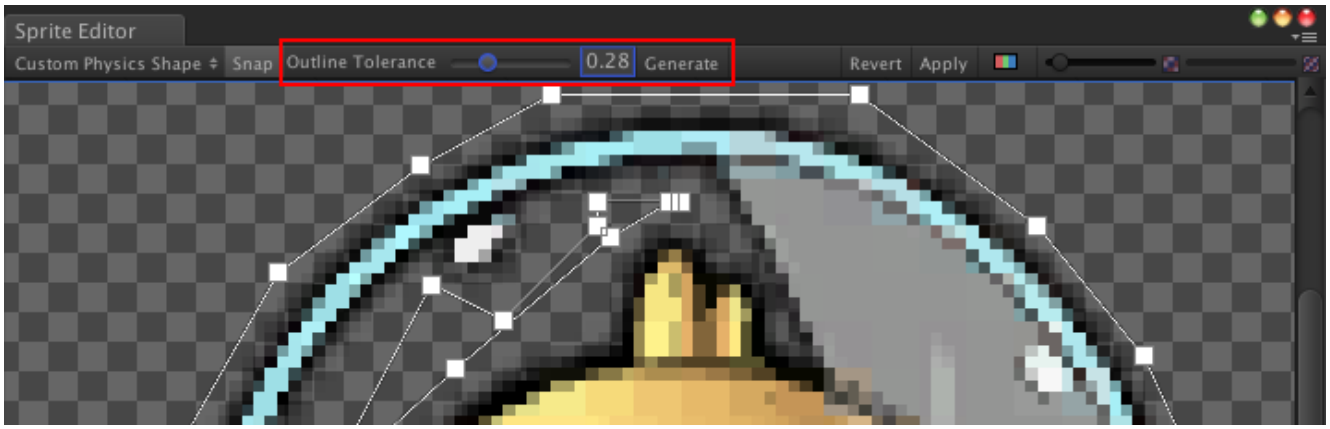
First open the Sprite Editor for your selected Sprite. Then, select **Custom Physics Shape** from the upper-left drop-down menu in the editor.



Then click **Generate** to create an outline of the Physics Shape. Unity generates an outline follows the shape of the original **Sprite** texture by default, and takes into account transparent areas as well.

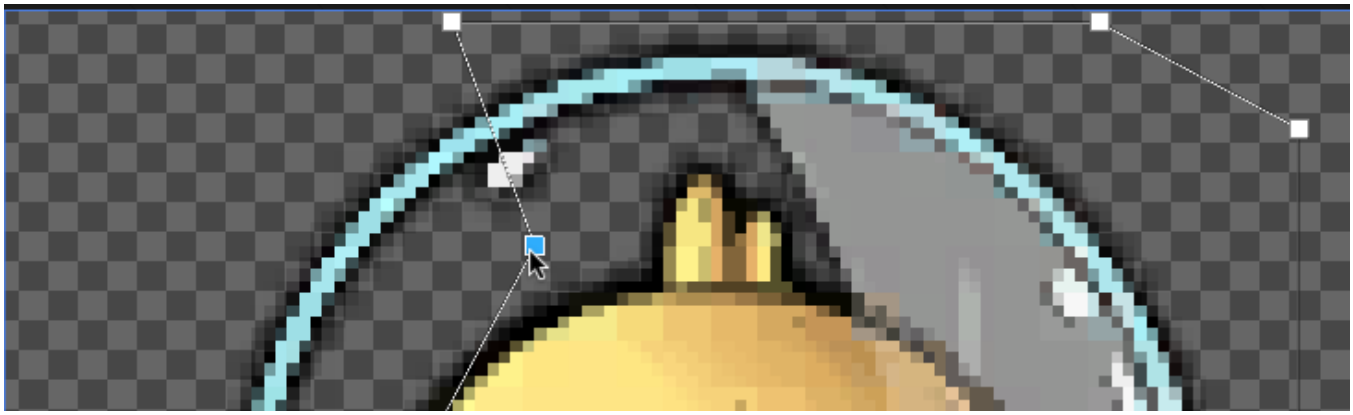
The generated outline and control points

Adjust the **Outline Tolerance** slider to refine the outline of the Physics Shape. After adjusting the **Outline Tolerance** value, click **Generate** to refresh the outline.



Outline Tolerance slider

Click and drag each control point to refine the outline of the Physics Shape. To remove a control point, select a control point and press the **Command+Del/Del** keys.



Moving a control point

When the mouse is hovering over the outline, a transparent control point appears along the edge. Click to create a new control point at that spot. Remove a control point by selecting it and pressing the **Del/Command+Del** keys.

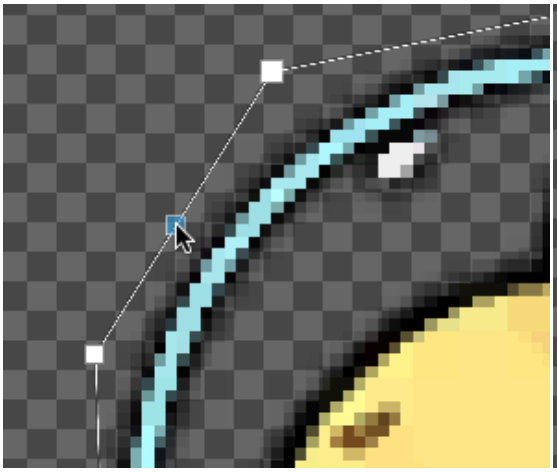


Fig.1: Transparent control point.

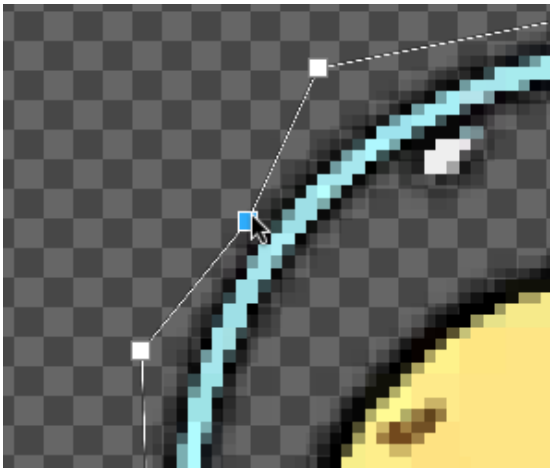
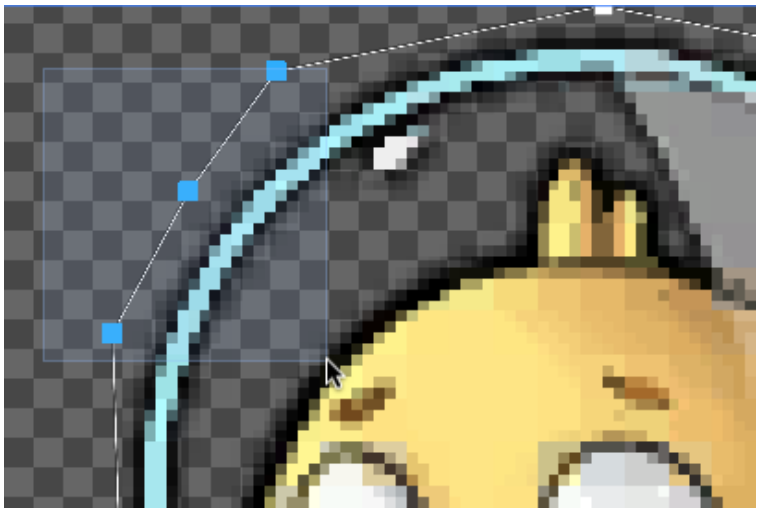


Fig.2: Click to create new control point.

Click and drag over an area to select multiple control points. You can position or delete them altogether while selected.



Selecting multiple control points

Holding the **Control/Ctrl** key allows you to select edges instead of their control points. Click on the highlighted edge to drag them into a new position.

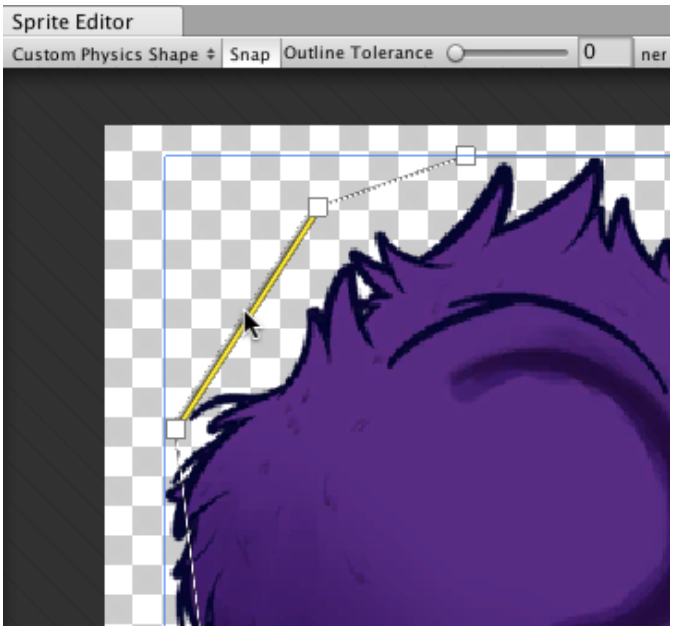


Fig.1: Select the edge of the outline.

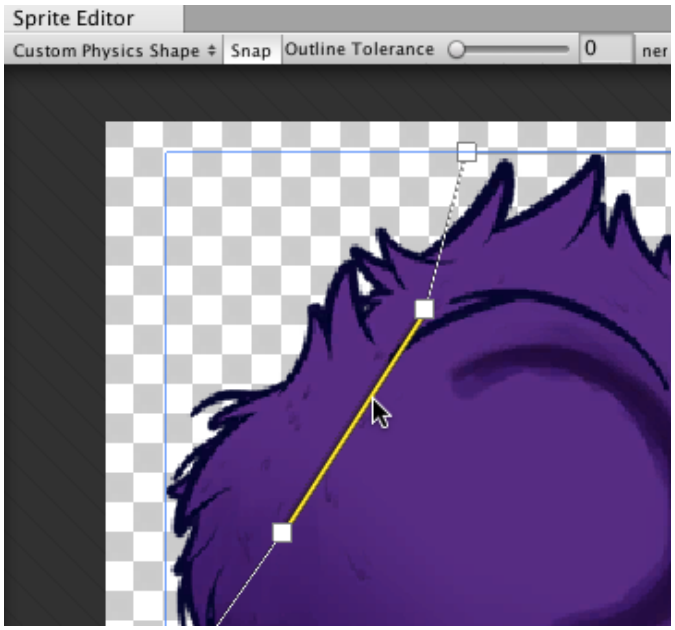


Fig.2: Drag and move the edge freely once selected.

Working With Multiple Outlines

A single Physics Shape can contain multiple separate outlines. This is useful if only specific areas of a Sprite need a Collider 2D Mesh for **collision**. For example, you might want a character to only respond to collisions on specific areas of its Sprite for damage as part of the game mechanics.

Click and drag over any empty space in the Sprite Editor window to create a new rectangular outline with 4 control points. Repeat this step to create additional outlines. You can refine each outline in the same way you would for a single Physics Shape outline.

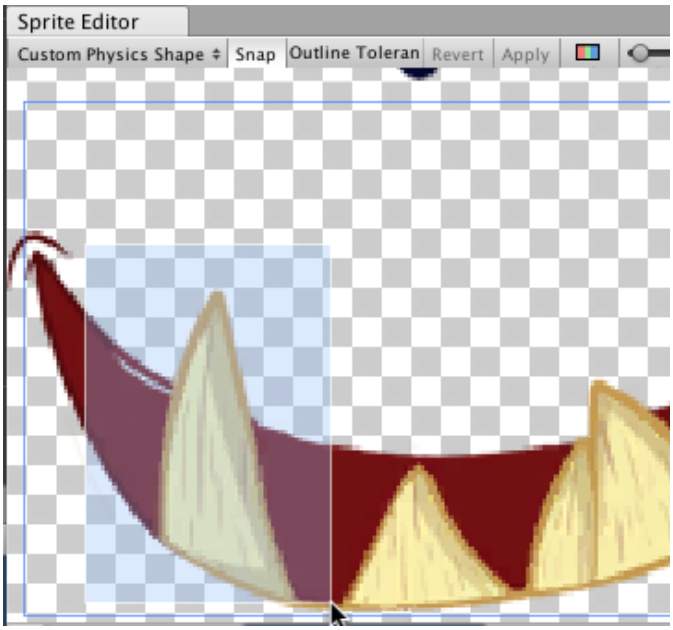


Fig. 1: Click and drag to create 4-point box.

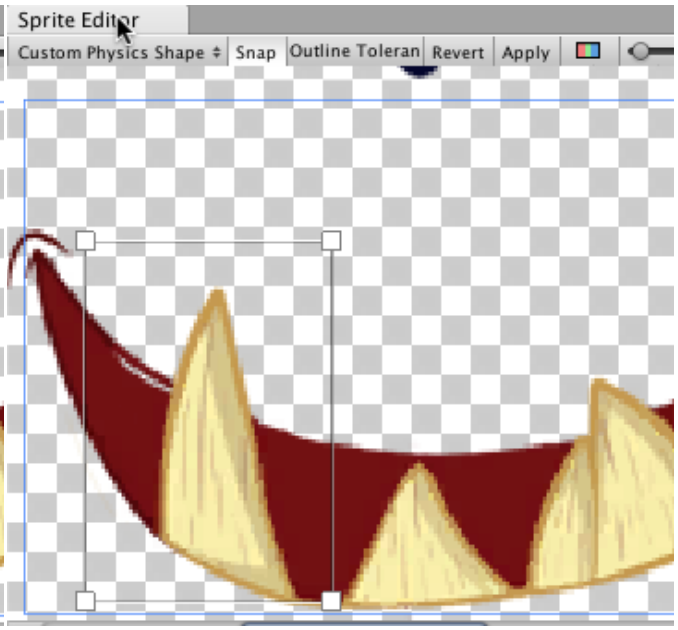


Fig. 2: Box physics shape with 4 control points.

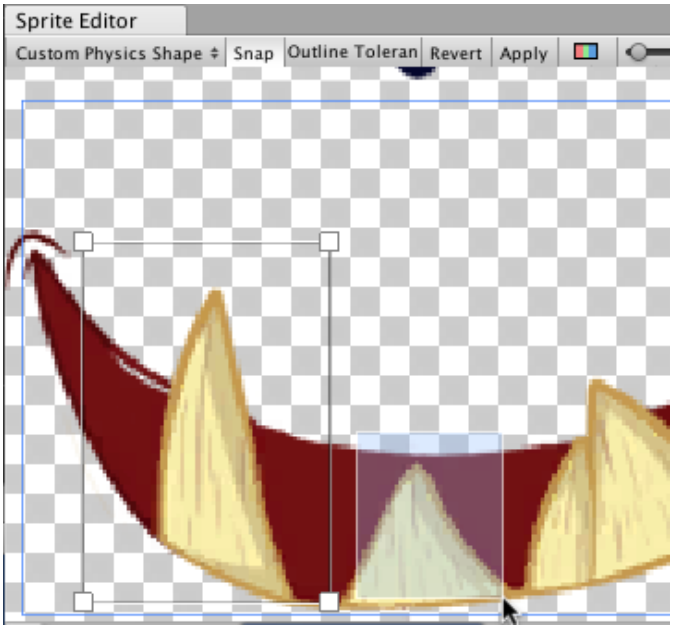


Fig. 3: Click and drag again for another box.

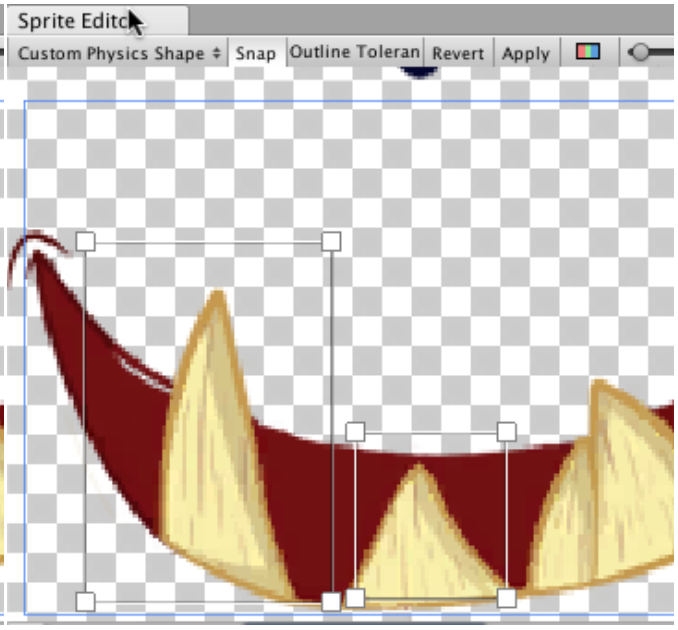
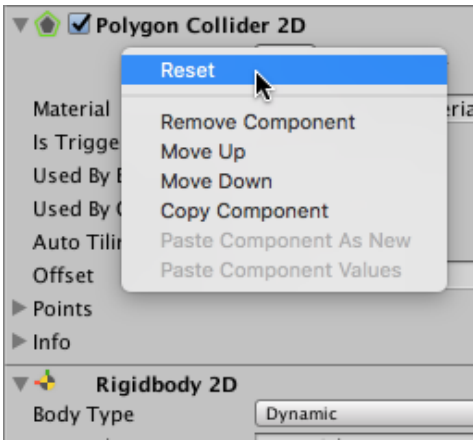


Fig. 4: Repeat to create more separate outlines.

Additional tips

If you have edited the outline of a Sprite that existing **GameObjects** already refer to, right-click the title of the **Collider 2D** component and select **Reset**. This updates the shape of the Collider 2D Meshes.



2018–05–24 Page published with editorial review

Sprite Packer

Leave feedback

When designing **sprite** graphics, it is convenient to work with a separate texture file for each character. However, a significant portion of a **sprite** texture will often be taken up by the empty space between the graphic elements and this space will result in wasted video memory at runtime. For optimal performance, it is best to pack graphics from several **sprite** textures tightly together within a single texture known as an atlas. Unity provides a **Sprite Packer** utility to automate the process of generating atlases from the individual **sprite** textures.

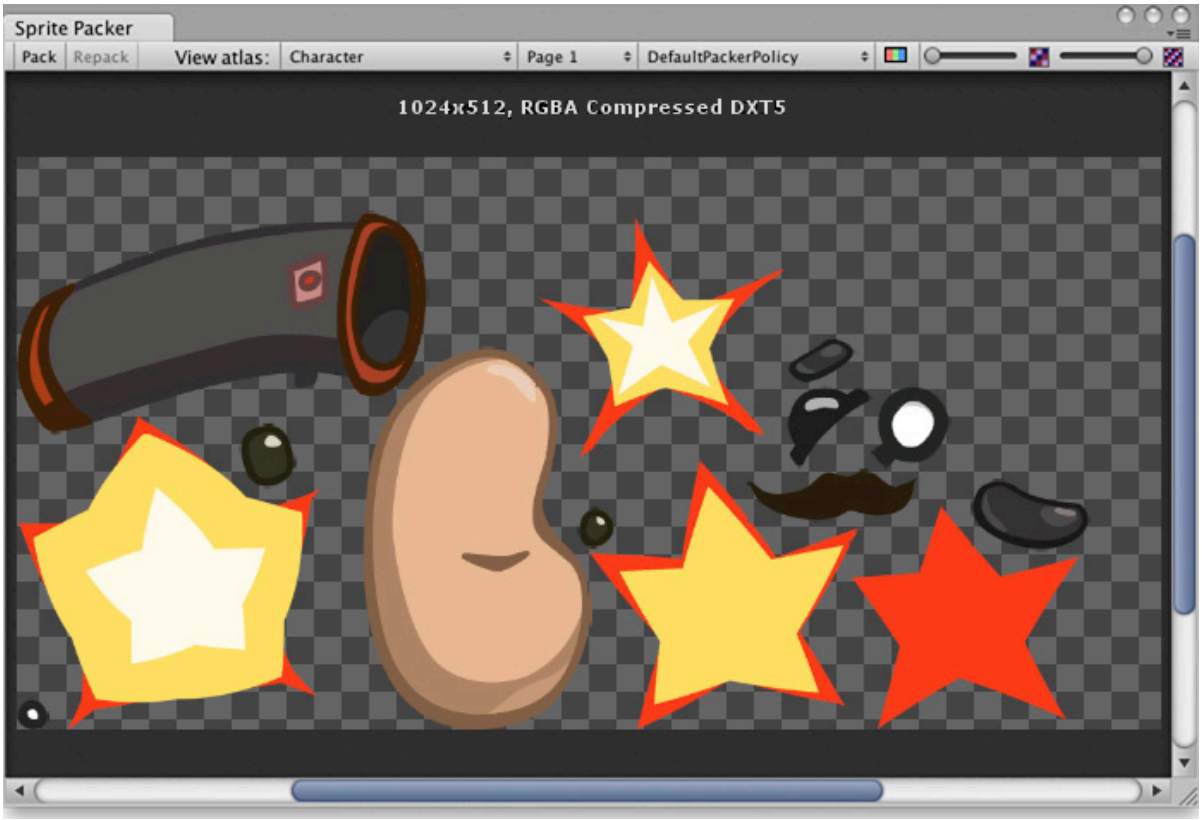
Unity handles the generation and use of sprite atlas textures behind the **scenes** so that the user needs to do no manual assignment. The atlas can optionally be packed on entering Play mode or during a build and the graphics for a sprite object will be obtained from the atlas once it is generated.

Users are required to specify a Packing Tag in the Texture Importer to enable packing for Sprites of that Texture.

Using the Sprite Packer

The Sprite Packer is disabled by default but you can configure it from the Editor settings (menu: **Edit** > **Project Settings** > **Editor**). The sprite packing mode can be changed from **Disabled** to **Enabled for Builds** (i.e. packing is used for builds but not Play mode) or **Always Enabled** (i.e. packing is enabled for both Play mode and builds).

If you open the Sprite Packer window (menu: **Window** > **2D** > **Sprite Packer**) and click the **Pack** button in the top-left corner, you will see the arrangement of the textures packed within the atlas.



If you select a sprite in the Project panel, this will also be highlighted to show its position in the atlas. The outline is actually the render **mesh** outline and it also defines the area used for tight packing.

The **toolbar** at the top of the Sprite Packer window has a number of controls that affect packing and viewing. The **Pack** buttons initiates the packing operation but will not force any update if the atlas hasn't changed since it was last packed. (A related **Repack** button will appear when you implement a custom packing policy as explained in **Customizing the Sprite Packer** below). The **View Atlas** and **Page #** menus allow you to choose which page of which atlas is shown in the window (a single atlas may be split into more than one "page" if there is not enough space for all sprites in the maximum texture size). The menu next to the page number selects which "packing policy" is used for the atlas (see below). At the right of the toolbar are two controls to zoom the view and to switch between color and alpha display for the atlas.

Packing Policy

The Sprite Packer uses a **packing policy** to decide how to assign sprites into atlases. It is possible to create your own packing policies (see below) but the **Default Packer Policy**, **Tight Packer Policy** and **Tight Rotate Enabled Sprite Packer Policy** options are always available. With these policies, the **Packing Tag** property in the Texture Importer directly selects the name of the atlas where the sprite will be packed and so all sprites with the same packing tag will be packed in the same atlas. Atlases are then further sorted by the texture import settings so that they match whatever the user sets for the source textures. Sprites with the same texture **compression** settings will be grouped into the same atlas where possible.

DefaultPackerPolicy will use rectangle packing by default unless "[TIGHT]" is specified in the **Packing Tag** (i.e. setting your packing tag to "[TIGHT]Character" will allow tight packing).
TightPackerPolicy will use tight packing by default if Sprite have tight meshes. If "[RECT]" is specified in the **Packing Tag**, rectangle packing will be done (i.e. setting your packing tag to "[RECT]UI_Elements" will force rect packing).
TightRotateEnabledSpritePackerPolicy will use tight packing by default if Sprite have tight meshes and will enable rotation of sprites. If "[RECT]" is specified in the **Packing Tag**, rectangle packing will be done (i.e. setting your packing tag to "[RECT]UI_Elements" will force rect packing).

Customizing the Sprite Packer

The **DefaultPackerPolicy** option is sufficient for most purposes but you can also implement your own custom packing policy if you need to. To do this, you need to implement the UnityEditor.Sprites.IPackerPolicy interface for a class in an editor script. This interface requires the following methods:

GetVersion - return the version value of your packer policy. Version should be bumped if modifications are done to the policy script and this policy is saved to **version control**.
OnGroupAtlases - implement your packing logic here. Define atlases on the PackerJob and assign Sprites from the given TextureImporters.
DefaultPackerPolicy uses rect packing (see SpritePackingMode) by default. This is useful if you're doing texture-space effects or would like to use a different mesh for **rendering** the Sprite. Custom policies can override this and instead use tight packing.

Repack button is only enabled when a custom policy is selected.
OnGroupAtlases will not be called unless TextureImporter metadata or the selected PackerPolicy version values change.
Use Repack button when working on your custom policy.
Sprites can be packed rotated with TightRotateEnabledSpritePackerPolicy automatically.
SpritePackingRotation is a reserved type for future Unity versions.

Other

Atlases are cached in Project\Library\AtlasCache.
Deleting this folder and then launching Unity will force atlases to be repacked. Unity must be closed when doing so.
Atlas cache is not loaded at start.
All textures must be checked when packing for the first time after Unity is restarted. This operation might take some time depending on the total number of textures in the project.
Only the required atlases are loaded.
Default maximum atlas size is 2048x2048.
When PackingTag is set, Texture will not be compressed so that the SpritePacker can grab original **pixel** values and then do compression on the atlas.

DefaultPackerPolicy

```
using System;
using System.Linq;
using UnityEngine;
using UnityEditor;
using System.Collections.Generic;
```

```
public class DefaultPackerPolicySample : UnityEditor.Sprites.IPackerPolicy
{
        protected class Entry
        {
            public Sprite            sprite;
        public UnityEditor.Sprites.AtlasSettings settings;
            public string            atlasName;
            public SpritePackingMode packingMode;
            public int               anisoLevel;
        }

        private const uint kDefaultPaddingPower = 3; // Good for base and two mip levels.

        public virtual int GetVersion() { return 1; }
        protected virtual string TagPrefix { get { return "[TIGHT]"; } }
        protected virtual bool AllowTightWhenTagged { get { return true; } }
        protected virtual bool AllowRotationFlipping { get { return false; } }

    public static bool IsCompressedFormat(TextureFormat fmt)
    {
        if (fmt >= TextureFormat.DXT1 && fmt <= TextureFormat.DXT5)
            return true;
        if (fmt >= TextureFormat.DXT1Crunched && fmt <= TextureFormat.DXT5Crunched)
            return true;
        if (fmt >= TextureFormat.PVRTC_RGB2 && fmt <= TextureFormat.PVRTC_RGBA4)
            return true;
        if (fmt == TextureFormat.ETC_RGB4)
            return true;
        if (fmt >= TextureFormat.ATC_RGB4 && fmt <= TextureFormat.ATC_RGBA8)
            return true;
        if (fmt >= TextureFormat.EAC_R && fmt <= TextureFormat.EAC_RG_SIGNED)
            return true;
        if (fmt >= TextureFormat.ETC2_RGB && fmt <= TextureFormat.ETC2_RGBA8)
            return true;
        if (fmt >= TextureFormat.ASTC_RGB_4x4 && fmt <= TextureFormat.ASTC_RGBA_12x12)
            return true;
        if (fmt >= TextureFormat.DXT1Crunched && fmt <= TextureFormat.DXT5Crunched)
            return true;
        return false;
    }

    public void OnGroupAtlases(BuildTarget target, UnityEditor.Sprites.PackerJob job, int[] te
        {
            List<Entry> entries = new List<Entry>();

            foreach (int instanceID in textureImporterInstanceIDs)
            {
                TextureImporter ti = EditorUtility.InstanceIDToObject(instanceID) as TextureIm

                TextureFormat desiredFormat;
                ColorSpace colorSpace;
                int compressionQuality;
                ti.ReadTextureImportInstructions(target, out desiredFormat, out colorSpace, ou
```

```
            TextureImporterSettings tis = new TextureImporterSettings();
            ti.ReadTextureSettings(tis);

        Sprite[] sprites =
            AssetDatabase.LoadAllAssetRepresentationsAtPath(ti.assetPath)
                .Select(x => x as Sprite)
                .Where(x => x != null)
                .ToArray();
            foreach (Sprite sprite in sprites)
            {
                Entry entry = new Entry();
                entry.sprite = sprite;
                entry.settings.format = desiredFormat;
                entry.settings.colorSpace = colorSpace;
                // Use Compression Quality for Grouping later only for Compressed Formats.
            entry.settings.compressionQuality = IsCompressedFormat(desiredFormat) ? compre
            entry.settings.filterMode = Enum.IsDefined(typeof(FilterMode), ti.filterMode)
                ? ti.filterMode
                : FilterMode.Bilinear;
                entry.settings.maxWidth = 2048;
                entry.settings.maxHeight = 2048;
                entry.settings.generateMipMaps = ti.mipmapEnabled;
                entry.settings.enableRotation = AllowRotationFlipping;
                if (ti.mipmapEnabled)
                    entry.settings.paddingPower = kDefaultPaddingPower;
                else
                    entry.settings.paddingPower = (uint)EditorSettings.spritePackerPadding
                #if ENABLE_ANDROID_ATLAS_ETC1_COMPRESSION
                    entry.settings.allowsAlphaSplitting = ti.GetAllowsAlphaSplitting ();
                #endif //ENABLE_ANDROID_ATLAS_ETC1_COMPRESSION

                entry.atlasName = ParseAtlasName(ti.spritePackingTag);
                entry.packingMode = GetPackingMode(ti.spritePackingTag, tis.spriteMeshType
                entry.anisoLevel = ti.anisoLevel;

                entries.Add(entry);
            }

            Resources.UnloadAsset(ti);
        }

        // First split sprites into groups based on atlas name
        var atlasGroups =
            from e in entries
            group e by e.atlasName;
        foreach (var atlasGroup in atlasGroups)
        {
            int page = 0;
            // Then split those groups into smaller groups based on texture settings
            var settingsGroups =
                from t in atlasGroup
                group t by t.settings;
            foreach (var settingsGroup in settingsGroups)
```

```
                {
                    string atlasName = atlasGroup.Key;
                    if (settingsGroups.Count() > 1)
                        atlasName += string.Format(" (Group {0})", page);

                UnityEditor.Sprites.AtlasSettings settings = settingsGroup.Key;
                    settings.anisoLevel = 1;
                    // Use the highest aniso level from all entries in this atlas
                    if (settings.generateMipMaps)
                        foreach (Entry entry in settingsGroup)
                            if (entry.anisoLevel > settings.anisoLevel)
                                settings.anisoLevel = entry.anisoLevel;

                    job.AddAtlas(atlasName, settings);
                    foreach (Entry entry in settingsGroup)
                    {
                        job.AssignToAtlas(atlasName, entry.sprite, entry.packingMode, SpritePa
                    }

                    ++page;
                }
            }
        }

        protected bool IsTagPrefixed(string packingTag)
        {
            packingTag = packingTag.Trim();
            if (packingTag.Length < TagPrefix.Length)
                return false;
            return (packingTag.Substring(0, TagPrefix.Length) == TagPrefix);
        }

        private string ParseAtlasName(string packingTag)
        {
            string name = packingTag.Trim();
            if (IsTagPrefixed(name))
                name = name.Substring(TagPrefix.Length).Trim();
            return (name.Length == 0) ? "(unnamed)" : name;
        }

        private SpritePackingMode GetPackingMode(string packingTag, SpriteMeshType meshType)
        {
            if (meshType == SpriteMeshType.Tight)
                if (IsTagPrefixed(packingTag) == AllowTightWhenTagged)
                    return SpritePackingMode.Tight;
            return SpritePackingMode.Rectangle;
        }
    }
```

TightPackerPolicy

```
using System;
using System.Linq;
using UnityEngine;
using UnityEditor;
using UnityEditor.Sprites;
using System.Collections.Generic;

// TightPackerPolicy will tightly pack non-rectangle Sprites unless their packing tag contains
class TightPackerPolicySample : DefaultPackerPolicySample
{
        protected override string TagPrefix { get { return "[RECT]"; } }
        protected override bool AllowTightWhenTagged { get { return false; } }
        protected override bool AllowRotationFlipping { get { return false; } }
}
```

TightRotateEnabledSpritePackerPolicy

```
using System;
using System.Linq;
using UnityEngine;
using UnityEditor;
using UnityEditor.Sprites;
using System.Collections.Generic;

// TightPackerPolicy will tightly pack non-rectangle Sprites unless their packing tag contains
class TightRotateEnabledSpritePackerPolicySample : DefaultPackerPolicySample
{
        protected override string TagPrefix { get { return "[RECT]"; } }
        protected override bool AllowTightWhenTagged { get { return false; } }
        protected override bool AllowRotationFlipping { get { return true; } }
}
```

Sorting Group

Leave feedback

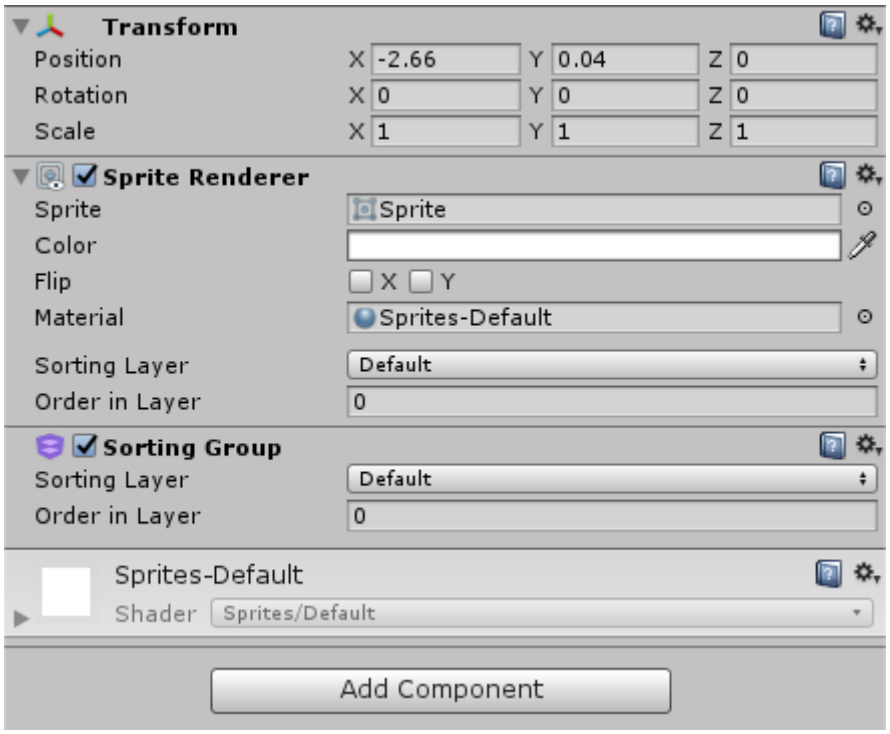
A Sorting Group is a component which alters the order in which Sprite Renderers do their rendering. It allows a group of Renderers which share a common root to be sorted together. Renderers in Unity are sorted by several criteria, including their order in the layer and their distance from the **Camera**.

Setting up a Sorting Group

To use the Sorting Group component, add it to the GameObject's root (the parent GameObject of all the GameObjects you want to apply group sorting to). Select the GameObject's root, then in the main menu select **Component** > **Rendering** > **Sorting Group**.

A Sorting Group does not have any visual representation in the **Scene view**. It can be added to an empty GameObject, which might be useful if you have many **GameObjects** to apply group sorting to at once.

A Sorting Group is not dependent on any other Renderers, and any Renderers that are attached to that GameObject and its descendants are rendered together.



The Sorting Group component

Property	Function
Sorting Layer	Use the drop-down to select the Layer used to define this **Sprite**'s overlay priority during rendering.
Order in Layer	Set the overlay priority of this Sprite within its layer. Lower numbers are rendered first, and subsequent numbers overlay those below.

Sorting a Sorting Group

Unity uses the concept of sorting layers to allow you to divide sprites into groups for overlay priority. Sorting Groups with a **Sorting Layer** lower in the order are overlaid by those in a higher **Sorting Layer**.

Sometimes, two or more objects in the same **Sorting Layer** can overlap (for example, two player characters in a side scrolling game, as shown in the example below). The **Order in Layer** property can be used to apply

consistent priorities to Sorting Groups in the same layer. As with **Sorting Layer**, lower numbers are rendered first, and are obscured by Sorting Groups with higher layer numbers, which are rendered later. See the documentation on Tags and Layers for details on editing **Sorting Layers**.

The descendants of the Sorting Group are sorted against other descendants of closest or next Sorting Group (depending on whether sorting is by distance or Order in Layer). In other words, the Sorting Group creates a local sorting space for its descendants only. This allows each of the Renderers inside the group to be sorted using the **Sorting Layer** and **Order in Layer**, but locally to the containing Sorting Group.

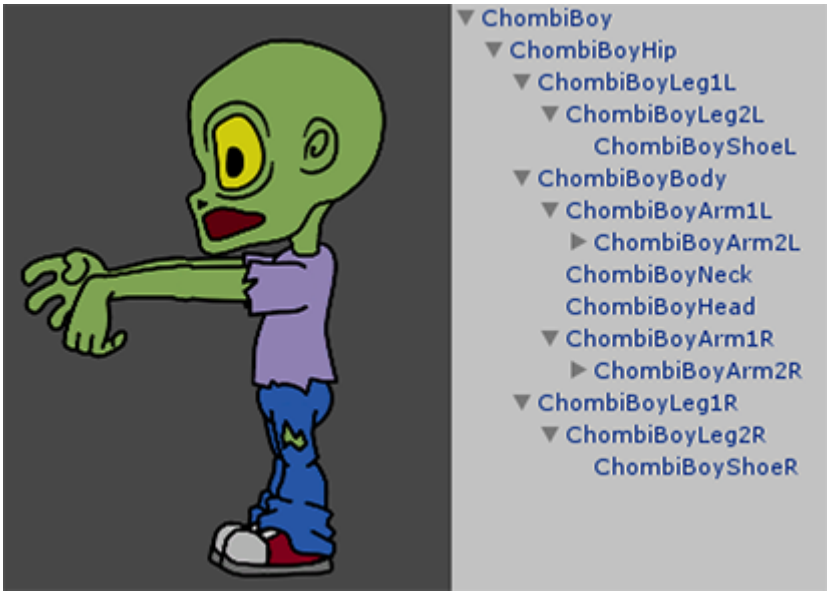
Nested Sorting Group

A nested Sorting Group is sorted against other Renderers in the same group.

However, GameObjects in the Hierarchy that do not have a Sorting Group are rendered together as a single layer, and Renderers are still sorted based on their **Sorting Layer** and **Order in Layer**.

Example of how to use Sorting Groups

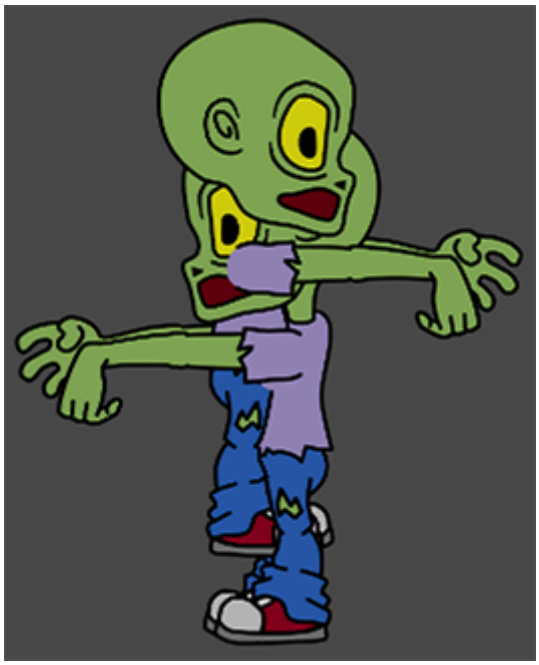
Sorting Groups are commonly used in 2D games with complex characters, made up of several Sprites. This example uses a 2D character with multiple Renderers in a Hierarchy.



A character made up of several Sprites in a single Sorting Layer, using multiple **Order in Layers** to sort its body parts

This character is in a single Sorting Layer, and uses multiple **Order in Layers** to sort its body parts. Unity then saves the character as a Prefab, and clones it multiple times during gameplay.

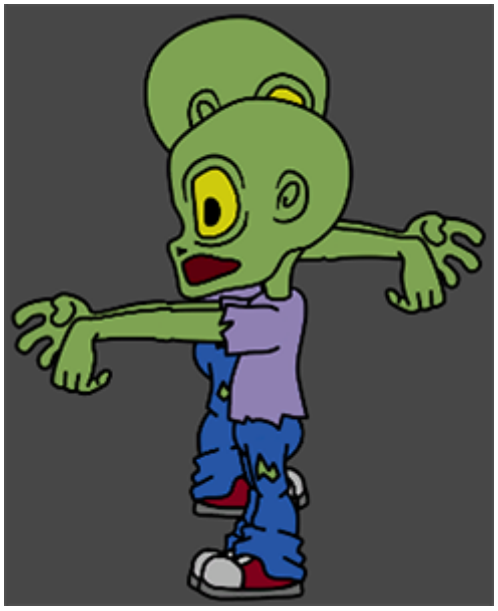
When cloned, the body parts overlap each other because they are on the same layers, as seen below.



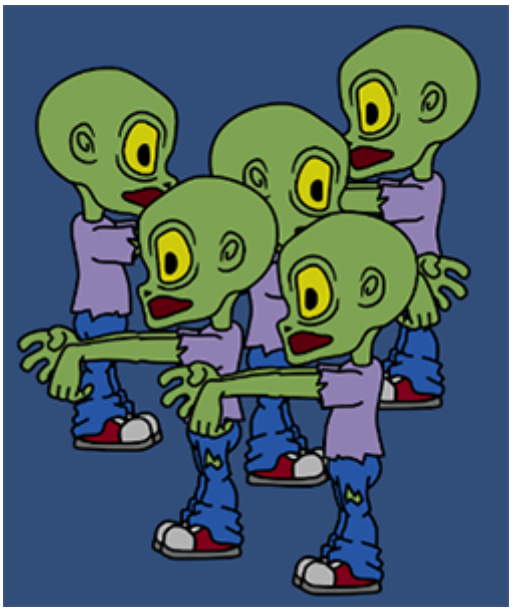
The body parts of two characters overlap, because they share the same layers

The desired outcome is to have all the Renderers of one character render together, and then be followed by the next character. This gives the visual effect of passing each other, with one appearing closer to the camera than the other, rather than both of them appearing to blend together.

A Sorting Group component added to the root of the character ensures that the body parts no longer overlap and mix together.



The Sorting Group component sorts each character as a group, preventing the issue of overlapping body parts



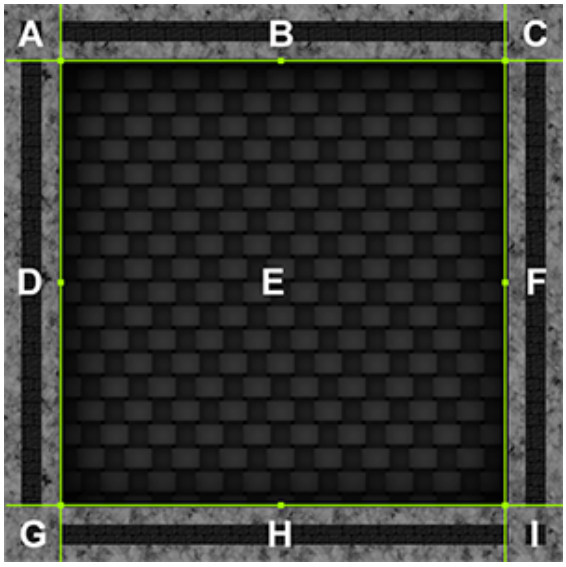
Multiple characters being sorted using a Sorting Group component on each character

9-slicing Sprites

Leave feedback

9-slicing is a 2D technique which allows you to reuse an image at various sizes without needing to prepare multiple Assets. It involves splitting the image into nine portions, so that when you re-size the Sprite, the different portions scale or tile (that is, repeat in a grid formation) in different ways to keep the Sprite in proportion. This is useful when creating patterns or Textures, such as walls or floors in a 2D environment.

This is an example of a 9-sliced Sprite, split into nine sections. Each section is labelled with a letter from A to I.



The following points describe what happens when you change the dimensions of the image:

The four corners (A, C, G and I) do not change in size.

The B and H sections stretch or tile horizontally.

The D and F sections stretch or tile vertically.

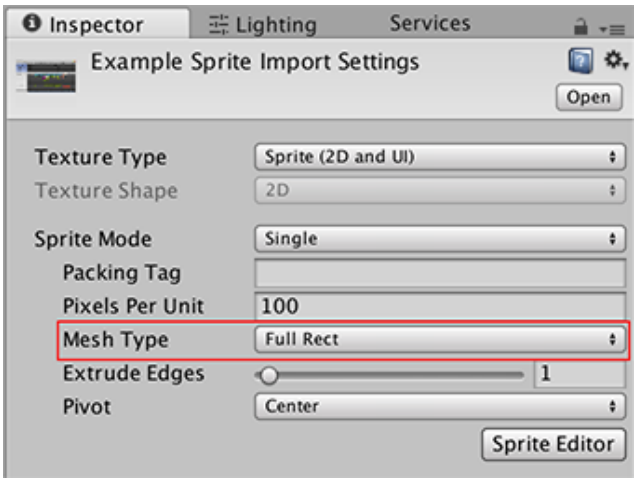
The E section stretches or tiles both horizontally and vertically.

This page describes how to set 9-slicing up, and which settings to apply depending on whether you want to stretch or tile the areas shown above.

Setting up your Sprite for 9-slicing

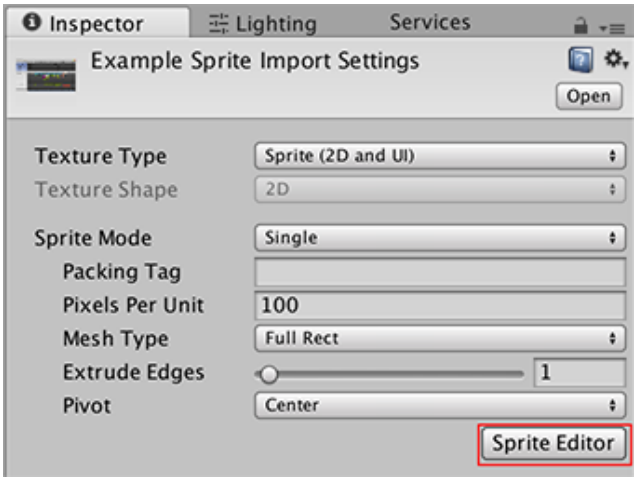
Before you 9-slice a Sprite, you need to ensure the Sprite is set up properly.

First, you need to make sure the **Mesh Type** is set to **Full Rect**. To apply this, select the Sprite, then in the Inspector window click the **Mesh Type** drop-down and select **Full Rect**. If the **Mesh Type** is set to **Tight**, 9-slicing might not work correctly, because of how the Sprite Renderer generates and renders the Sprite when it is set up for 9-slicing.



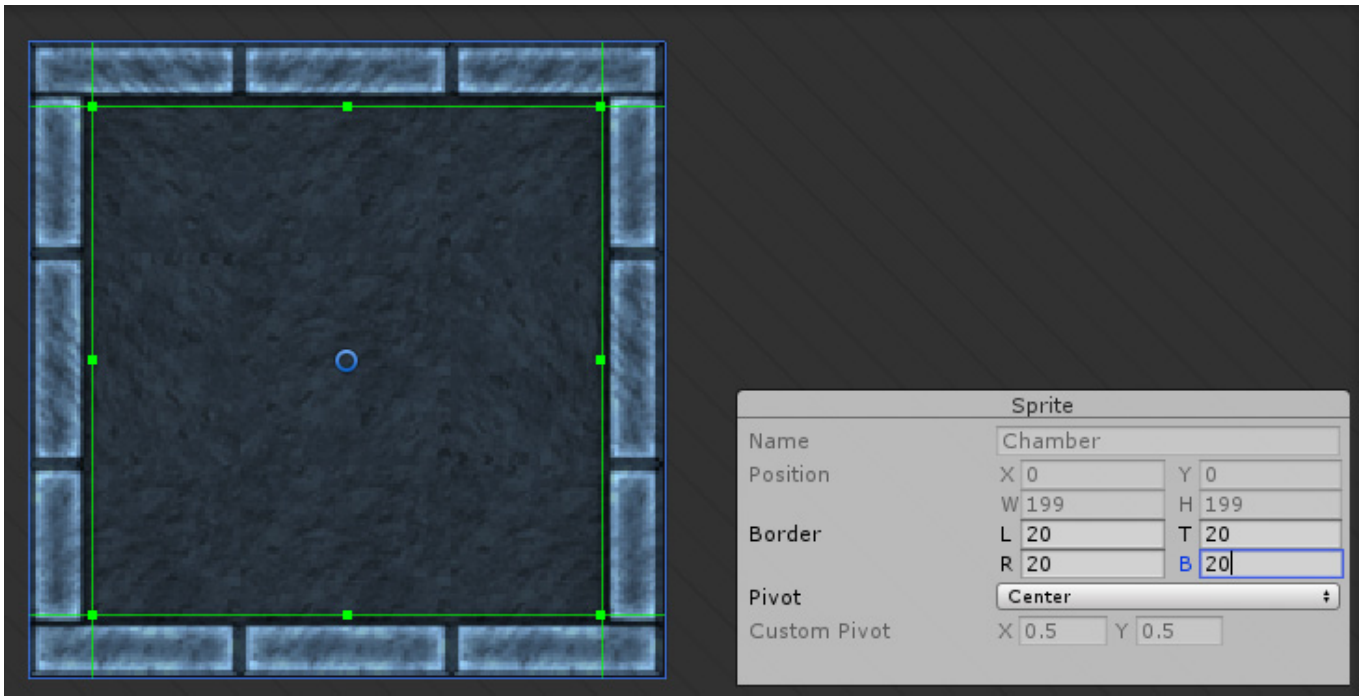
The Sprite's Inspector window. **Mesh Type** is highlighted in the red box.

Next, you need to define the borders of the Sprite via the **Sprite Editor** window. To do this, select the Sprite, then in the Inspector window click the **Sprite Editor** button.



The Sprite's Inspector window. The **Sprite Editor** button is highlighted in the red box. See documentation on Sprites for information on all of the properties in the Sprite Import Settings.

Use the Sprite Editor window to define the borders of the Sprite (that is, where you want to define the tiled areas, such as the walls of a floor tile). To do this, use the Sprite control panel's **L**, **R**, **T**, and **B** fields (left, right, top, and bottom, respectively). Alternatively, click and drag the green dots at the top, bottom, and sides.

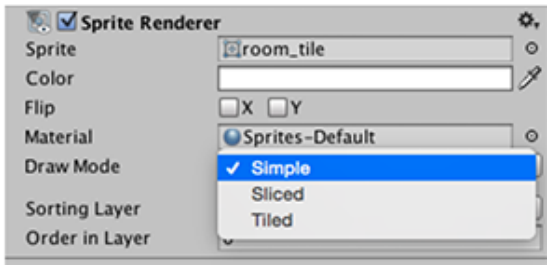


Defining the borders of the Sprite in the Sprite Editor window

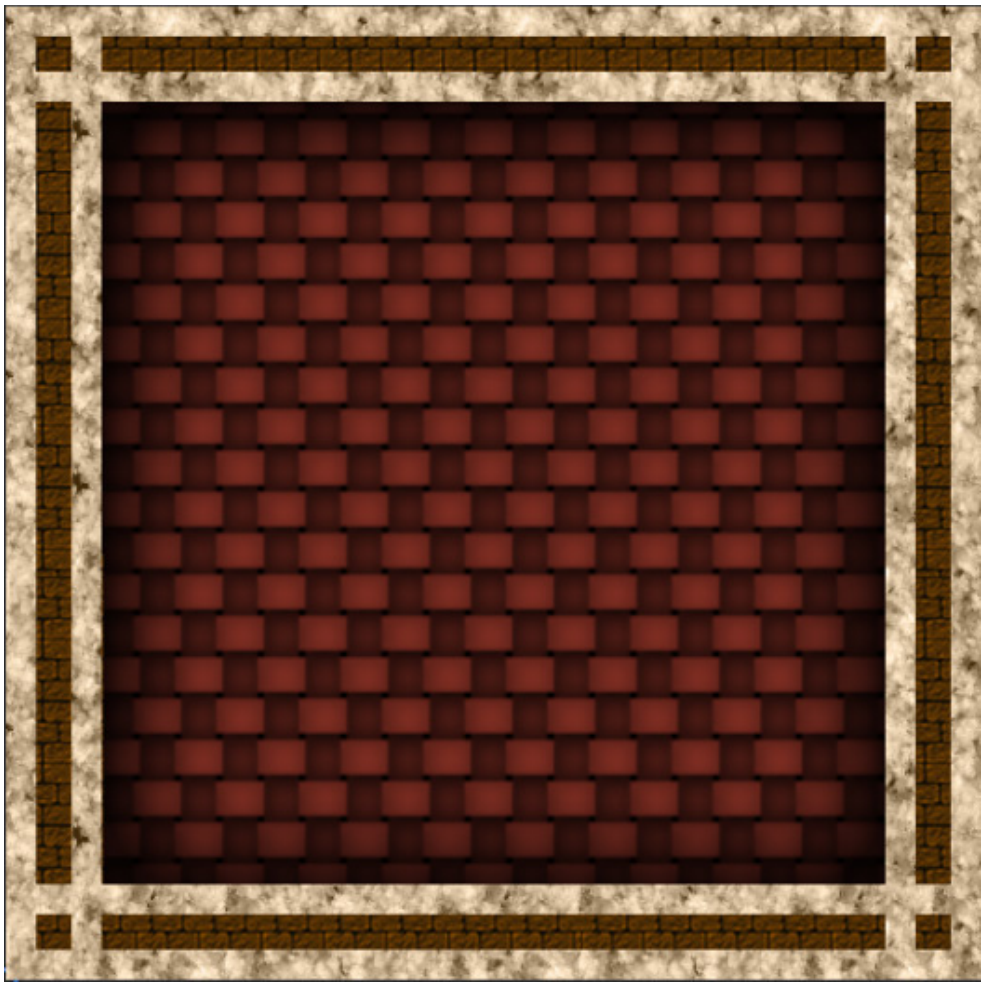
Click **Apply** in the Sprite Editor window's top bar. Close the Sprite Editor window, and drag the Sprite from the Project window into the **Scene** view to begin working on it.

9-slicing your Sprite

Select the Sprite in the Scene view or the Hierarchy window. In the Inspector window, navigate to the Sprite Renderer **component** and change the **Draw Mode** property.

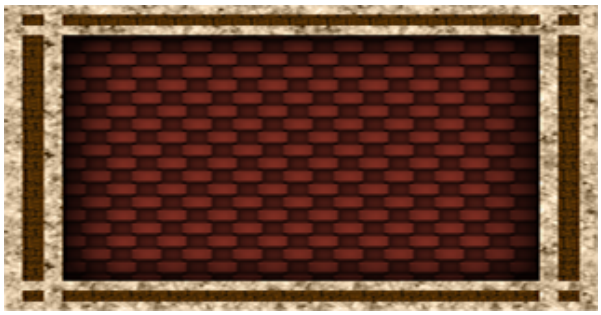


It is set to **Simple** by default; to apply 9-slicing, set it to either **Sliced** or **Tiled** depending on the behavior you want. The follow sections explain how each one behaves, using this Sprite:



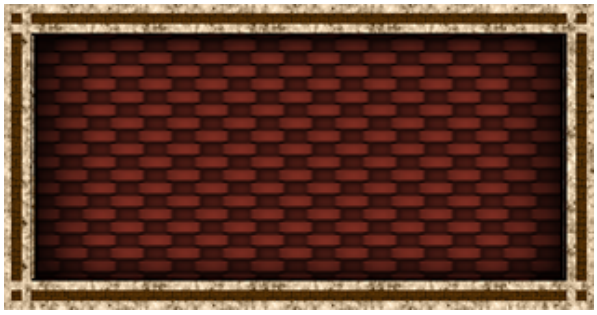
The original Sprite used for the examples shown below

Simple



This is the default Sprite Renderer behaviour. The image scales in all directions when its dimensions change. **Simple** is not used for 9-slicing.

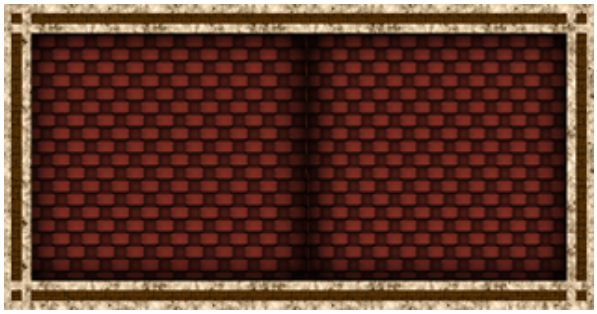
Sliced



In **Sliced** mode, the corners stay the same size, the top and bottom of the Sprite stretch horizontally, the sides of the Sprite stretch vertically, and the centre of the Sprite stretches horizontally and vertically to fit the Sprite's size.

When a Sprite's **Draw Mode** is set to **Sliced**, you can choose to change the size using the **Size** property on the Sprite Renderer or the Rect Transform Tool. You can still scale the Sprite using the Transform properties or the Transform Tool; however, the Transform scales the Sprite without applying the 9-slicing.

Tiled



In **Tiled** mode, the sprite stays the same size, and does not scale. Instead, the top and bottom of the Sprite repeat horizontally, the sides repeat vertically, and the centre of the Sprite repeats in a tile formation to fit the Sprite's size.

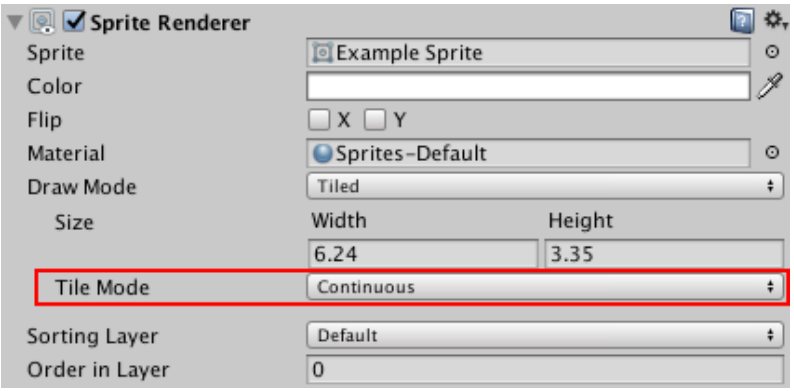
When a Sprite's **Draw Mode** is set to **Sliced**, you can choose to change the size using the **Size** property on the Sprite Renderer or the Rect Transform Tool. You can still scale the Sprite using the Transform properties or the Transform Tool; however, the Transform scales the Sprite without applying the 9-slicing.

When you set the **Draw Mode** to **Tiled**, an additional property called **Tile Mode** appears. See the next section in this page for more information on how **Tile Mode** works.

See documentation on the Sprite Renderer for full details on all of the component's properties.

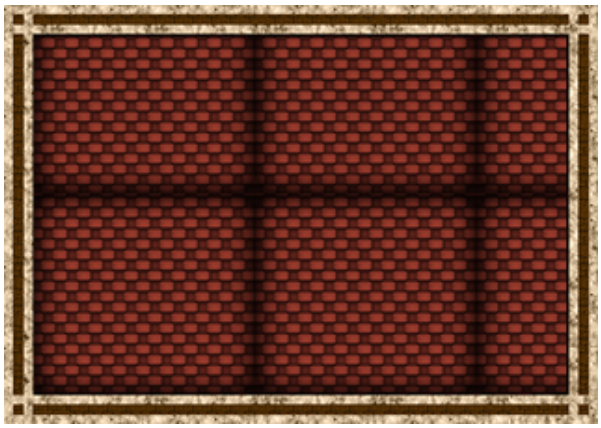
Tile Mode

When the **Draw Mode** is set to **Tiled**, use the **Tile Mode** property to control how the sections repeat when the dimensions of the Sprite change.



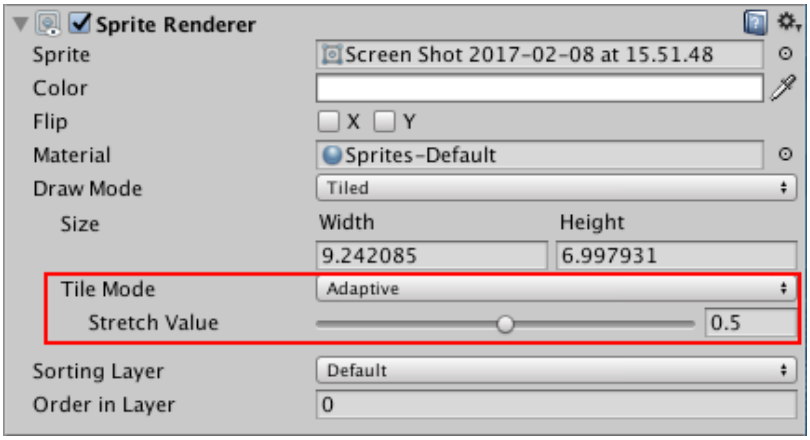
Continuous

Tile Mode is set to **Continuous** by default. When the size of the Sprite changes, the repeating sections repeat evenly in the Sprite.



Adaptive

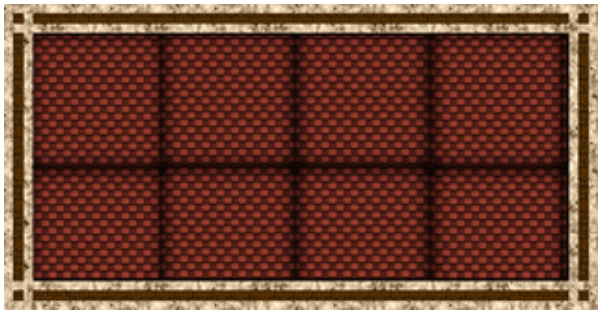
When **Tile Mode** is set to **Adaptive**, the repeating sections only repeat when the dimensions of the Sprite reach the **Stretch Value**.



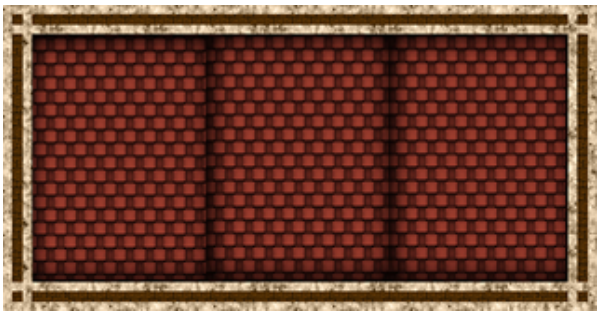
Use the **Stretch Value** slider to set the value between **0** and **1**. Note that **1** represents an image resized to twice its original dimensions, so if the **Stretch Value** is set at **1**, the section repeats when the image is stretched to twice its original size.

To demonstrate this, the following images show the difference between an image with the same dimension but a different **Stretch Value:**

Stretch Value 0.1:



Stretch Value 0.5:



9-slicing and Colliders

If your Sprite has a Collider2D attached, you need to ensure that when you change the Sprite's dimensions, the Collider2D changes with it.

Box Collider 2D and Polygon Collider 2D are the only Collider2D components in Unity that support 9-slicing. These two Collider2Ds have an **Auto Tiling** checkbox. To make sure the Collider2D components are set up for 9-slicing, select the Sprite you are applying it to, navigate to the Collider2D in the Inspector window, and tick the **Auto Tiling** checkbox. This enables automatic updates to the shape of the Collider2D, meaning that the shape is automatically readjusted when the Sprite's dimensions change. If you don't enable **Auto Tiling**, the Collider2D stays the same shape and size, even when the Sprite's dimensions change.

Limitations and known issues

The only two Collider2Ds that support 9-slicing are BoxCollider2D and PolygonCollider2D.

You cannot edit BoxCollider2D or PolygonCollider2D when the Sprite Renderer's **Draw Mode** is set to **Sliced** or **Tiled**. Editing in the Inspector window is disabled, and a warning appears to notify you that the Collider2D cannot be edited because it is driven by the Sprite Renderer component's tiling properties.

When the shape is regenerated in **Auto Tiling**, additional edges might appear within the Collider2D's shape. This may have an effect on **collisions**.

Sprite Masks

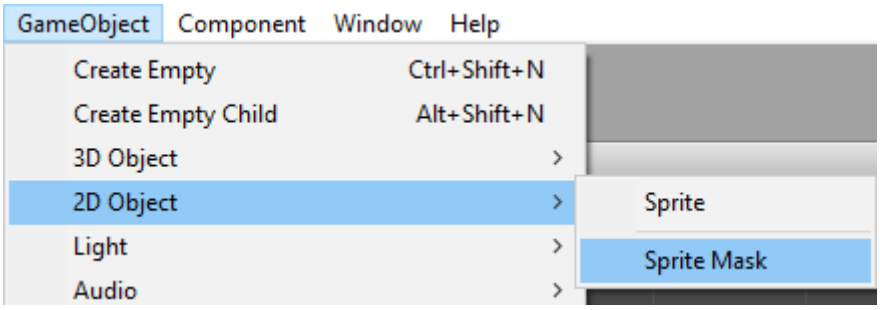
Leave feedback

SWITCH TO SCRIPTING

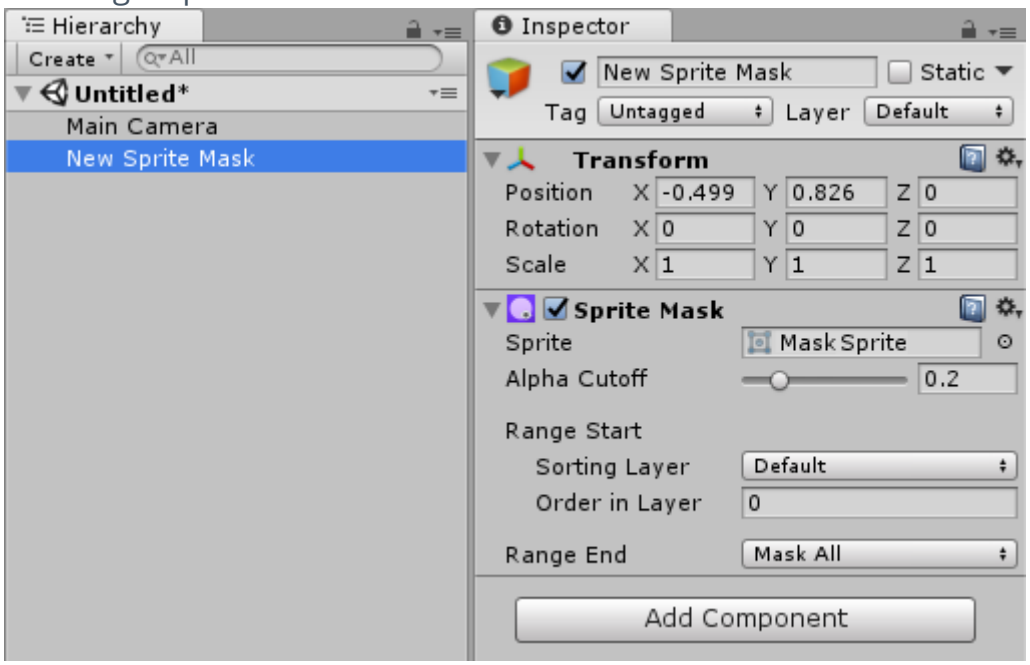
Sprite Masks are used to either hide or reveal parts of a Sprite or group of **Sprites**. The **Sprite Mask** only affects objects using the **Sprite Renderer Component**.

Creating a Sprite Mask

To create a Sprite Mask select from the main menu GameObject > 2D Object > Sprite Mask.

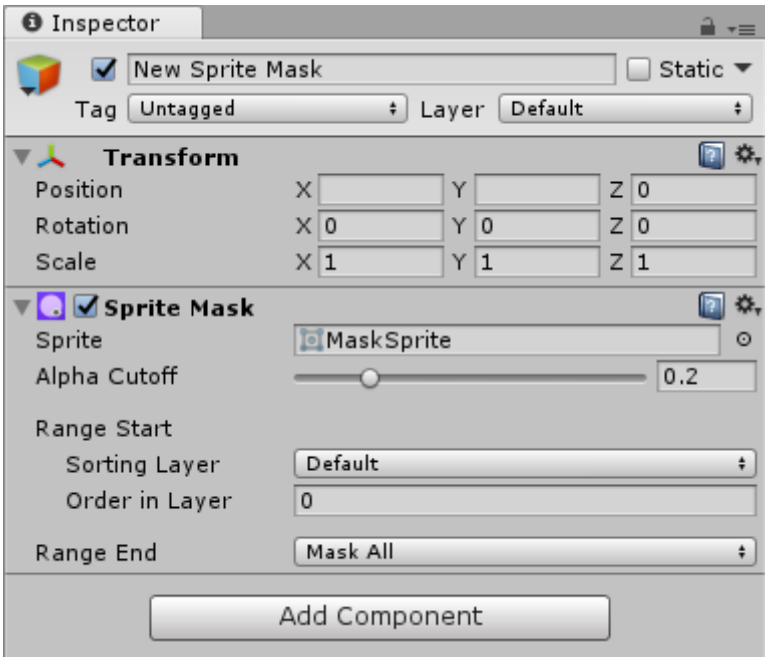


Creating a Sprite Mask from the menu



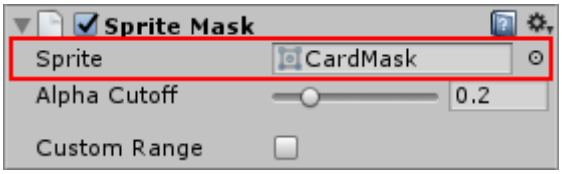
A new Sprite Mask GameObject is created in the Scene

Properties



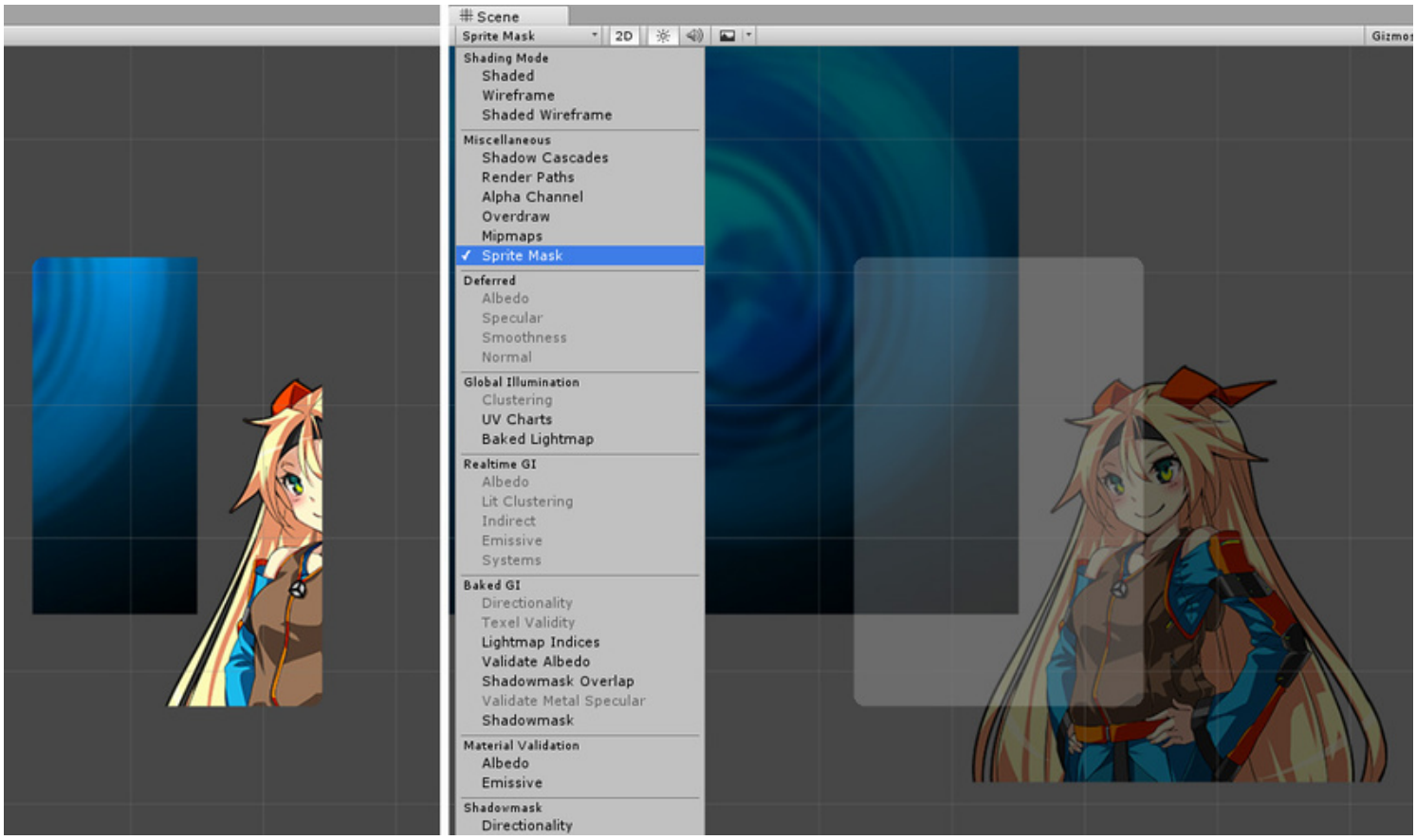
Property	Function
Sprite	The sprite to be used as a mask.
Alpha Cutoff	If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for which areas will be shown. You change this cutoff by adjusting the Alpha Cutoff slider.
Range Start	The Range Start is the Sorting Layer which the mask starts masking from.
Sorting Layer	The Sorting Layer for the mask.
Order in Layer	The order within the Sorting Layer.
Range End	
Mask All	By default the mask will affect all sorting layers behind it (lower sorting order).
Custom	The range end can be set to a custom Sorting Layer and Order in Layer.

Using Sprite Masks



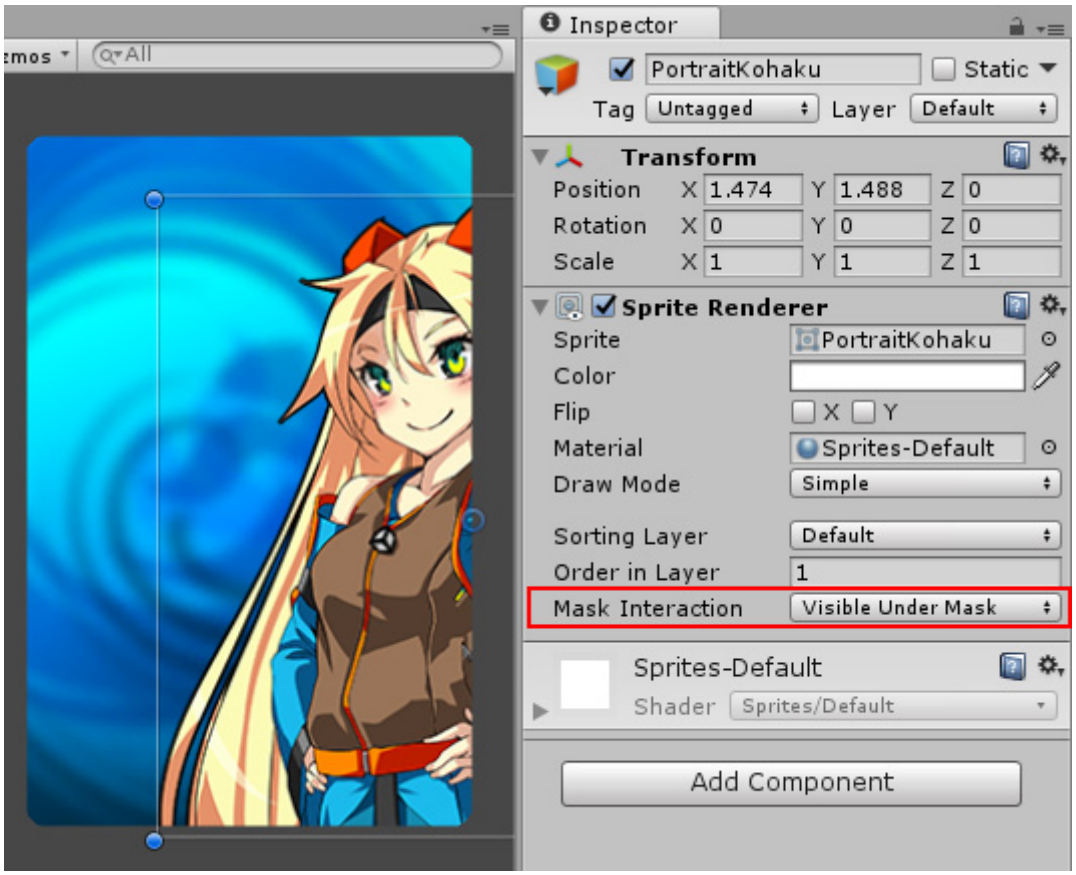
The Sprite to be used as a mask needs to be assigned to the Sprite Mask Component

The Sprite Mask **GameObject** itself will not be visible in **scene**, only the resulting interactions with Sprites. To view Sprite Masks in the **scene**, select the Sprite Mask option in the **Scene** Menu.



Scene view with Sprite Mask view turned on in the scene

Sprite Masks are always in effect. Sprites to be affected by a Sprite Mask need to have their Mask Interaction set in the Sprite Renderer.

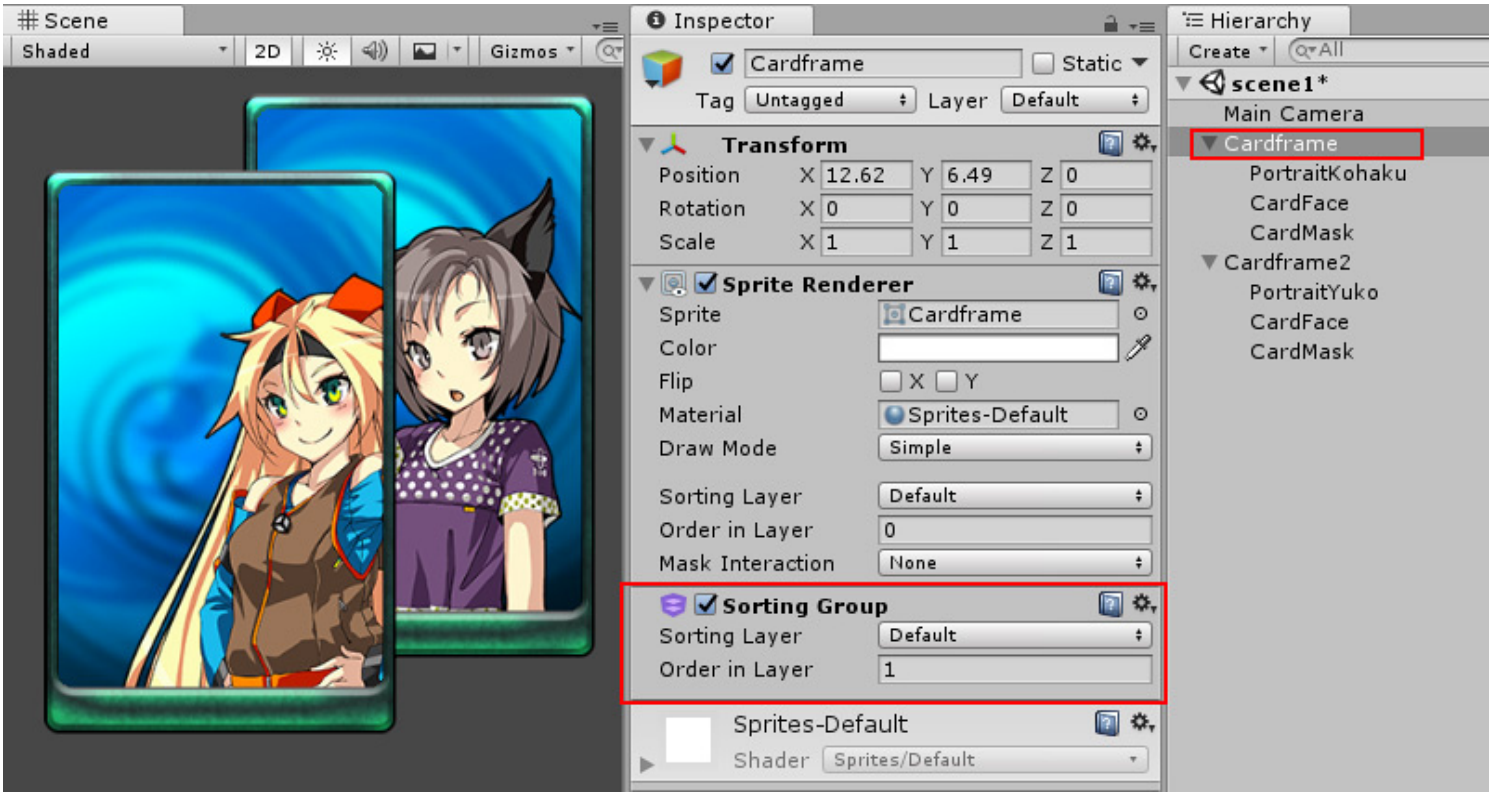


The character sprites Mask Interaction is set to Visible Under Mask thus only parts of the sprite that are in the mask area are visible

By default a Sprite Mask will affect any sprite in the scene that has their Mask Interaction set to Visible or Not Visible Under Mask. Quite often we want the mask to only affect a particular sprite or a group of sprites.

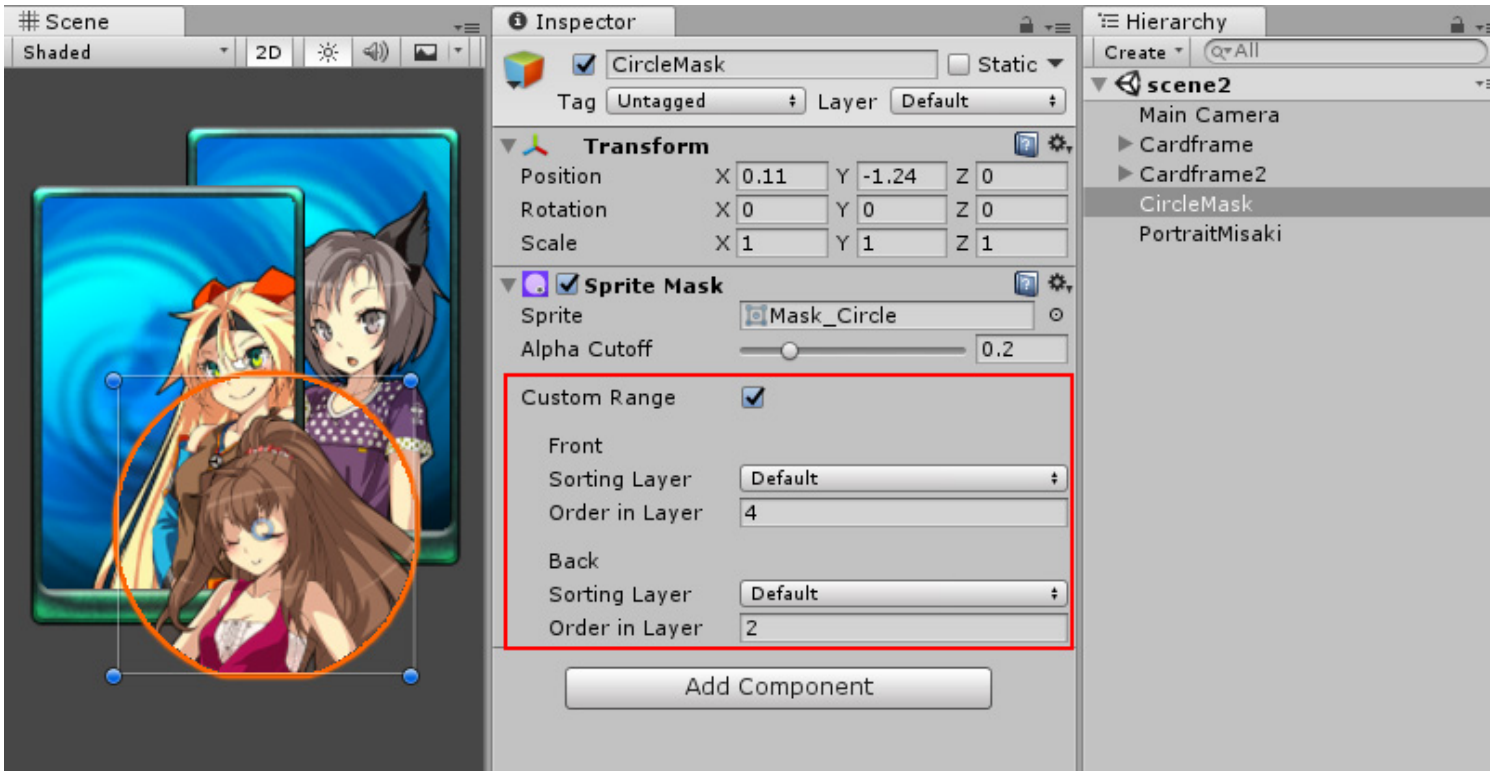
The character sprites are interacting with masks on both the cards

One method of ensuring the mask is interacting with particular sprites is to use a Sorting Group Component.



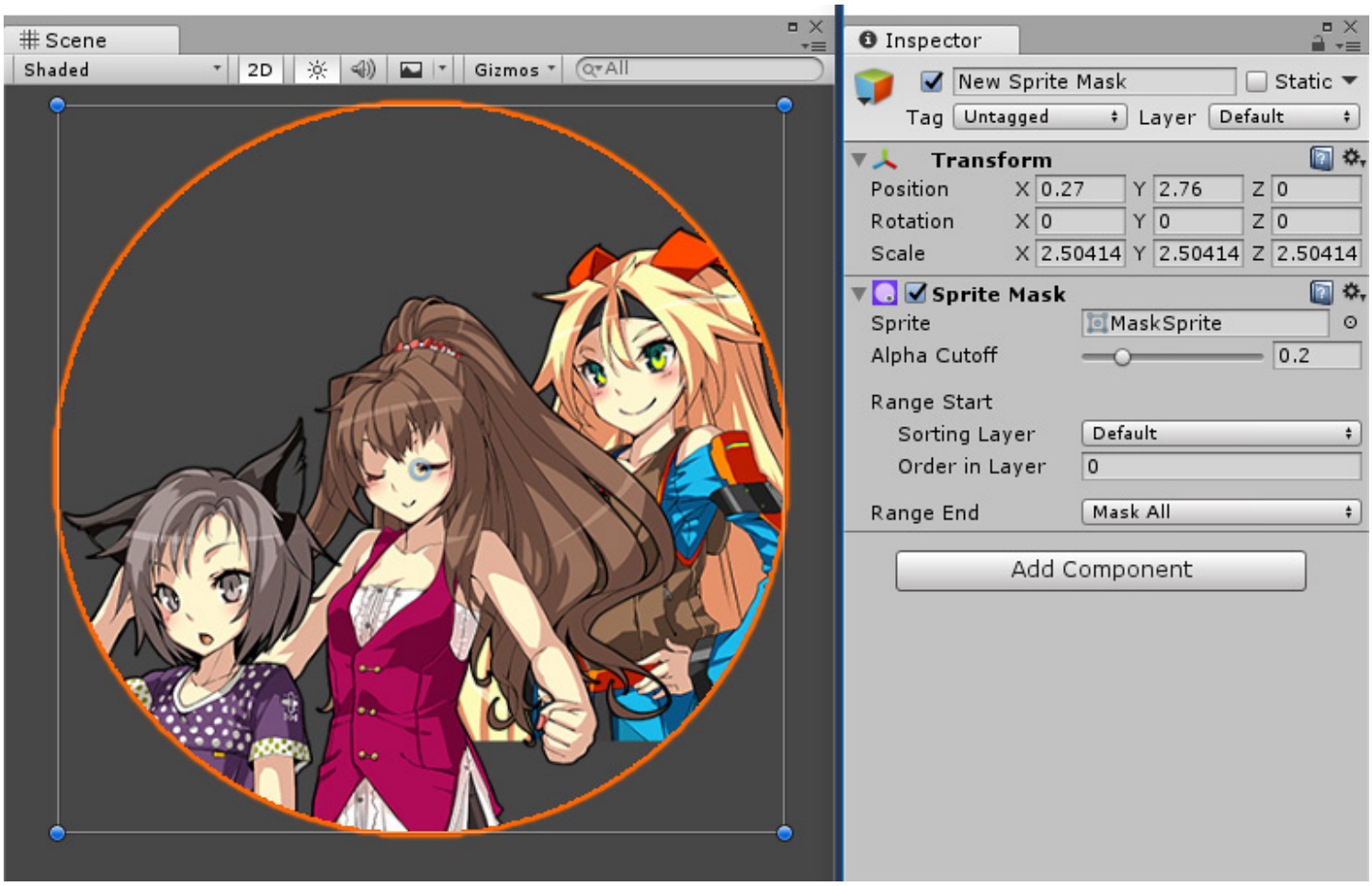
Sorting Group Component added to the Parent GameObject ensures the mask will affect only children of that Sorting Group

An alternative method of controlling the effect of the mask is to use the Custom Range Settings of a Sprite Mask.



A Sprite Mask with a Custom Range setting ensures the mask will affect only Sprites in the specified Sorting Layer or Order in Layer range

The Range Start and Range End provides the ability to selectively mask sprites based on their Sorting Layer or Order in Layer.



2017–05–26 Page published with no editorial review

New feature in Unity 2017.1

Sprite Atlas

Leave feedback

When designing **Sprites** and other graphic elements graphics for a 2D project, many separate texture files eventually become included in the project. However, a significant portion of Sprite textures may be taken up by empty spaces between each of these graphic elements. These additional empty spaces result in wasted video memory at runtime.

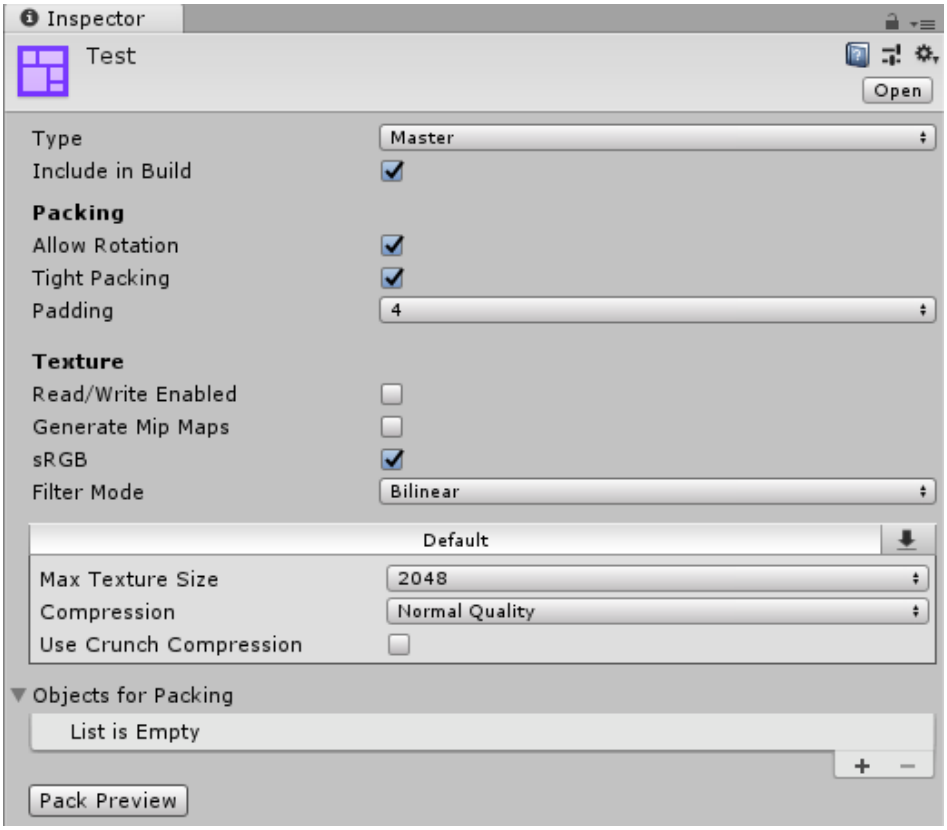
For optimal performance, it is recommended to pack graphics from several Sprite textures tightly together within a single **Asset**, known as the Sprite Atlas.

Creating a Sprite Atlas

Create a Sprite Atlas from the main menu (menu: **Asset > Create > Sprite Atlas**)

Once created, the Sprite Atlas is placed in the Project's Asset folder like other Assets, but with its own unique file extension (*.spriteatlas).

Properties



The Sprite Atlas Asset provides a set of unified Texture settings for all packed Sprite Textures packed in it. Regardless of the original texture settings of the packed Sprites, the resulting single Atlas texture only reflects the settings defined in the Sprite Atlas Asset properties.

Property	Function
Type	Sets the type of Atlas to Master or Variant.
Include in Build	Check to include the Atlas Asset in the build. Note that unchecking this option causes any packed Assets to not be rendered during Play Mode.
Allow Rotation	Allow Sprites to be rotated for packing.
Tight Packing	Use the Sprite outlines to fit them during packing instead of rectangle **mesh** outlines.
Padding	Amount of extra padding between packed Sprite textures.

Property	Function
Read/Write Enabled	Set to true to allow texture data to be readable/writeable by **scripts**. Set to false to prevent **scripts** from reading/writing texture data.
Generate Mip Maps	Select this to enable mipmap generation.
sRGB	Textures are stored in gamma space.
Filter Mode	Select how the Texture is filtered. This overrides the packed Sprites' original texture settings.
Default Texture settings panel	Set default options (using Default), and then override them for a specific platform using the buttons along the top of the panel. https://docs.unity3d.com/Manual/class-TextureImporterOverride.html
Objects For Packing	Select objects to be packed into the Atlas. Eligible objects can be Folders, Textures, and Sprites.

Sprite Packer Mode Settings

The Sprite Packer mode settings are found under the Editor settings (menu: **Edit > Project Settings > Editor**). The select Mode determines how Textures from the Sprite Atlas are used within the Editor.

Enabled for Builds: Sprites are packed into the Sprite Atlas for builds only, and not for Play mode.
Always Enabled: Sprites are packed for Play Mode and Sprites resolve textures from the Sprite Atlas. However, Sprites resolve their textures from the original unpacked Texture during Edit Mode.
Always Enabled is enabled by default to allow the testing of packed Sprites loaded from an Asset Bundle during Play Mode.

Selecting Assets for Packing

The object types that can be assigned and packed into a Sprite Atlas Asset are folders, Texture2D, and Sprites.

Assigning a folder to the Sprite Atlas Asset includes all the Sprites and Texture2D within that folder and its subfolders for packing. Assigning a specific Sprite to the Atlas does *not* include all instances of that Sprite in the **Scene**. Assigning a Texture2D to the Atlas affects *all* instances of Sprites that references that specific Texture2D.

To select Assets for packing:

Select the Sprite Atlas Asset. The list under *Objects For Packing* shows the currently assigned Assets to the selected Atlas. add them by either adding new entry to the list or dragging and dropping them from the Project onto the list area in the **inspector**. You can add the folders, textures, sprites to the atlas.

Add Assets by selecting the + icon at the lower right of the list. The *Select Object* window appears and shows the available Assets in the current Project that can be packed into ths Sprite Atlas.

You can replace any assigned object in the list by selecting the circle icon to the right. This opens the *Select Object* window.

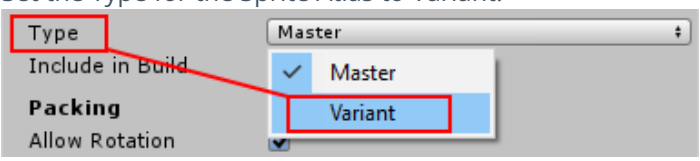
Click "Pack Preview" below the list to preview the packed Sprite Atlas texture that includes all objects in the list.

All Sprite Atlases are packed before entering Play Mode (unless *Include in Build* is unchecked)

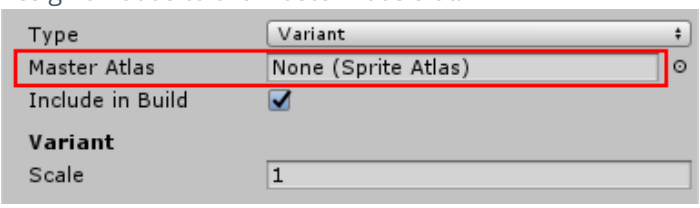
Creating a Sprite Atlas Variant

Declaring a Sprite Atlas as a Variant of another allows you to create a duplicate but resized version of the Master Atlas' texture. This is useful if you want to have both a Standard and High Definition version of the same texture for example.

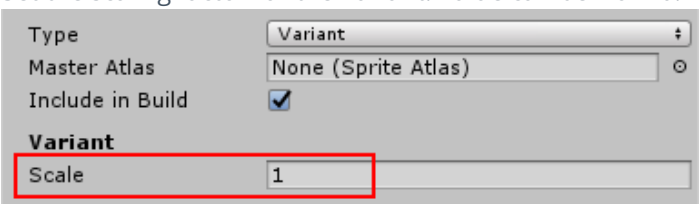
Set the Type for the Sprite Atlas to Variant.



Assign an atlas to the Master Atlas slot.



Set the scaling factor for the Variant. Value can be from 0.1 to 1.



To bind the Variant Sprite Atlas as the project's default instead of the Master Atlas, check the *Include in build* option in the Variant and uncheck that option in the Master. Checking both to be included in the build is not recommended.

Runtime Sprites enumeration

The Sprite Atlas Asset has a runtime representation which can be accessed during Runtime.

Create a custom **component** that takes a "SpriteAtlas" as a variable.

Assign any of your existing Sprite Atlas to the field.

Enter play mode or run the player.

Access the variable and notice you can now call the property ".GetSprites" to get the array of Sprites packed in this atlas.

Late Binding

A Sprite can be started in runtime as "packed; but not referencing an Atlas". It appears blank until an Atlas is bound to it. The *late binding* of the Sprite is useful if the source Atlas is not available during start-up, for example if the referenced Asset bundles are downloaded later.

Late Binding via callback

If a Sprite is packed into a Sprite Atlas that is not bound by default (e.g. *Include in build* is unchecked), then the Sprite appears invisible in the Scene.

User can listen to callback SpriteAtlasManager.atlasRequested.

This delegate method provides a tag of the Atlas which is to be bound, and a System.Action which takes in a SpriteAtlas Asset. The user is expected to load the Sprite Atlas Asset (e.g. by script references, Resources.load, Asset bundle etc.) and supply the Asset to the System.Action.

2018–07–30 Page amended with limited editorial review

2017–05–26 Page published with no editorial review

New in Unity 2017.1

Tilemap

Leave feedback

The **Tilemap** feature allows you to quickly create 2D levels using tiles and a grid overlay. It is comprised of a number of systems working together:

Tile Assets
Grid GameObjects
The **Tilemap Palette**
Custom **Brushes**
This section documents how to use each part of the Tilemap system.

2D extras in GitHub

Download example Brushes and Tiles from the 2D Extras GitHub repository.

https://github.com/Unity-Technologies/2d-extras

2017–09–06 Page published with limited editorial review

Tilemaps added in 2017.2

Tile Assets

Leave feedback

Typically, Tiles are actually Sprites that are arranged on a Tilemap. In Unity's implementation, we use an intermediary **Asset** that references the Sprite instead. This enables us to extend the Tile itself in many ways, creating a robust and flexible system for Tiles and **Tilemaps**.

Properties

Property	Function
Sprite	The sprite that is used by this tile Asset
Color	Used to tint the material
Collider Type	None, Sprite or Grid

2017–09–06 Page published with limited editorial review

Creating Tiles

Leave feedback

There are two ways to create Tiles. The first method is to directly create a Tile Asset. The other method is to automatically generate the Tiles from a selection of **Sprites**.

How to create a Tile Asset

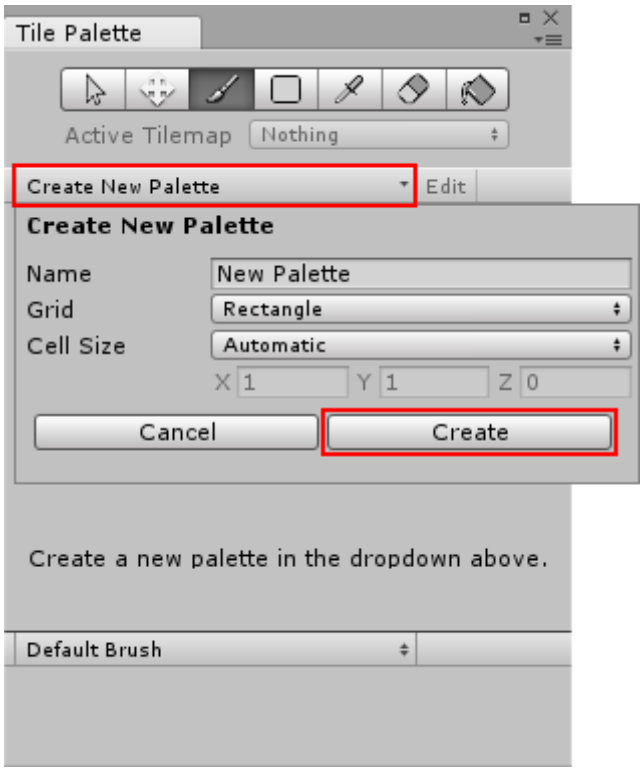
To create a Tile, go to the **Assets** menu and select **Create** > **Tile**. Then select where to save the new Tile Asset.

How to generate multiple Tile Assets

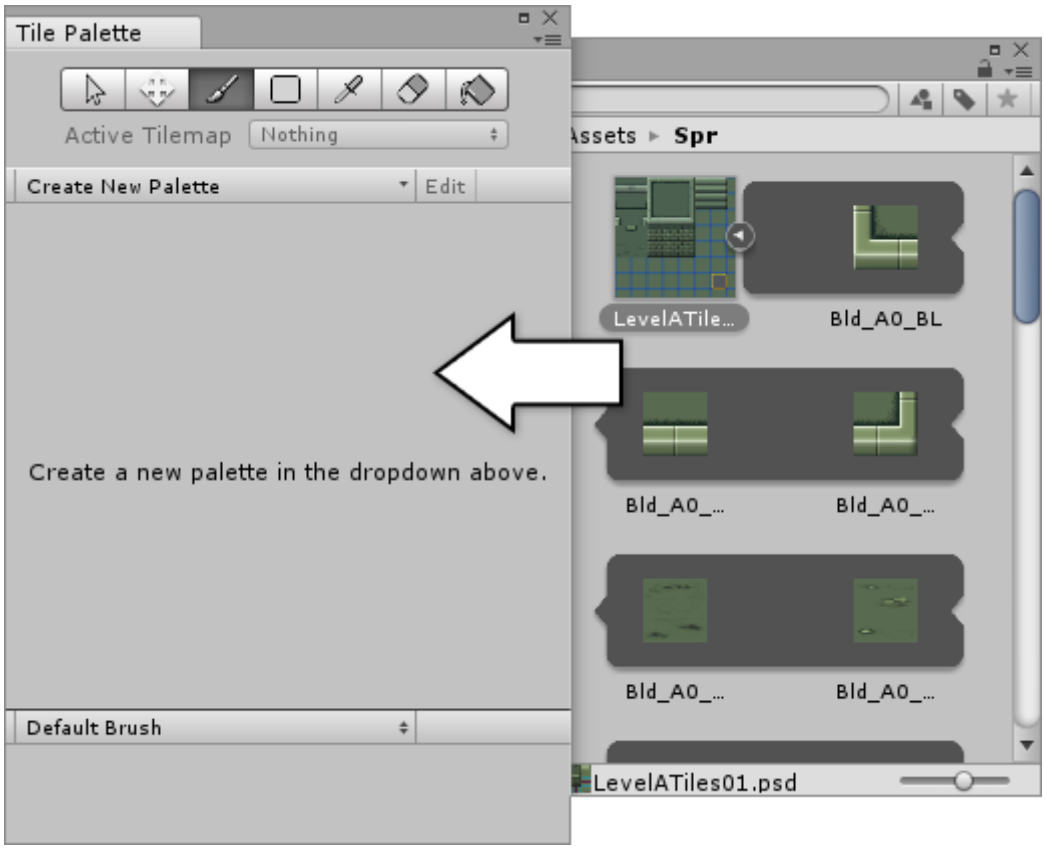
Automatic or multiple Tile creation requires a Palette to be loaded in the Tile Palette. If there is no Palette loaded you need to create a new Palette.

To create a new Palette, open the Tile Palette from **Window** > **2D** > **Tile Palette.**

Click the **Create New Palette** button in the Tile Palette. Provide a name for the Palette and click the **Create** button. Select a folder to save the Palette. The created Palette is automatically loaded.



Drag and drop Textures or Sprites from the *Assets* folder onto the Tile Palette. Choose where to save the new Tile Assets. New Tile Assets are generated in the selected folder and the Tiles are placed on the Tile Palette. Remember to save your project to save the Palette.



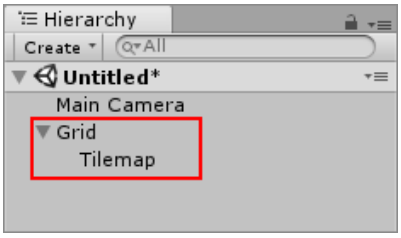
2018–04–12 Page published with limited editorial review

Creating Tilemaps

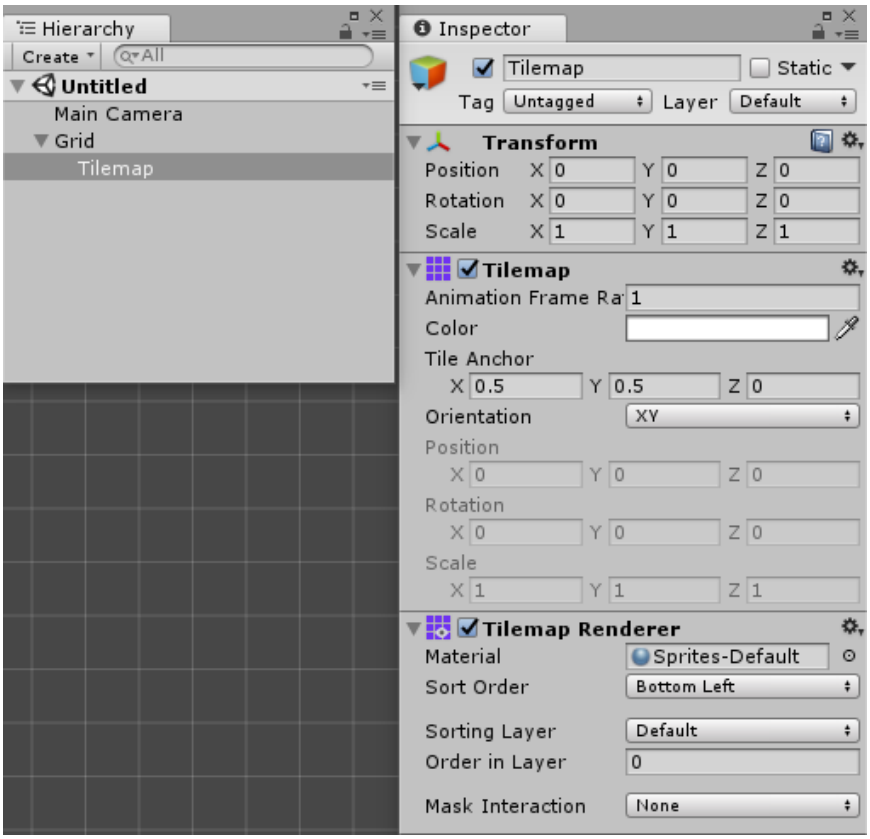
Leave feedback

From the **GameObject** menu, move to **2D Object** and select **Tilemap**.

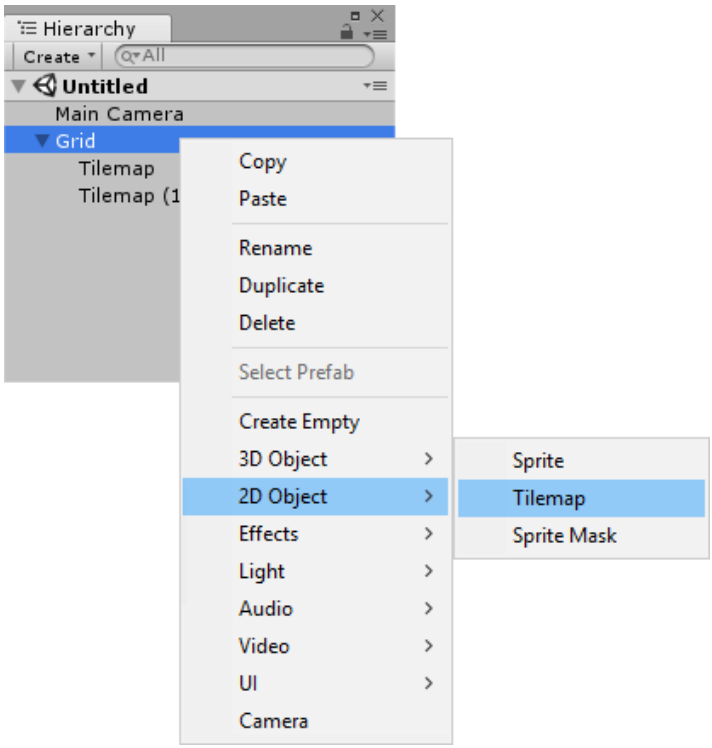
This creates a new GameObject with a child GameObject in the **scene**. The GameObject is called **Grid**. The Grid GameObject determines the layout of child Tilemaps.



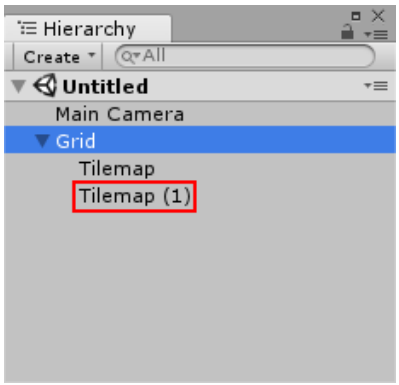
The child GameObject is called **Tilemap**. It is comprised of a Tilemap **component** and Tilemap Renderer **component**. The Tilemap GameObject is where Tiles are painted on.



To create additional Tilemaps to be used as “layers”, select the Grid GameObject or the Tilemap GameObject, and select **GameObject** > **2D Object** > **Tilemaps** in the menu or right-click on the selected GameObject and click on **2D Object** > **Tilemap**.

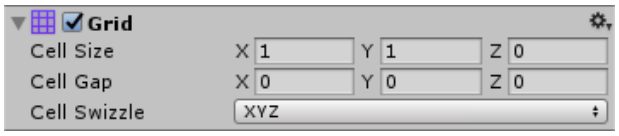


A new GameObject called **Tilemap (1)** is added into the hierarchy of the selected GameObject. You can paint Tiles on this new GameObject as well.



Adjusting the Grid for Tilemaps

Select the Grid GameObject. Adjust the values in the Grid component in the **inspector**.



Property	Function
Cell Size	Size of each cell in the Grid
Cell Gap	Size of the Gap between each cell in the Grid
Cell Swizzle	Swizzles the cell positions to other axes. For Example. In XZY mode, the Y and Z coordinates are swapped, so an input Y coordinate maps to Z instead and vice versa.

Changes in the Grid affect all child Layer GameObjects with the Tilemap, Tilemap Renderer and Tilemap **Collider** 2D components.

2017–09–06 Page published with limited editorial review

Tilemap Palette

Leave feedback

Alt + Left Mouse Button to pan

Middle Mouse Button to pan

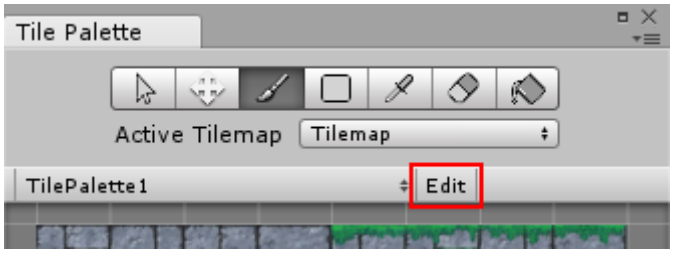
Mouse Wheel to zoom in or out

Left-click to select a Tile

Left-click drag to select multiple Tiles

Editing the Tilemap Palette

Select the desired palette from the drop-down menu. Click on the Edit button at the side of the palette selection menu.



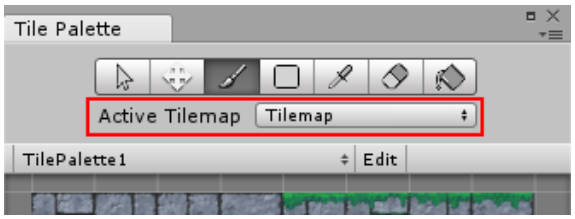
Adjust the Palette using the Tile Palette tools just like editing a **Tilemap** in the **scene**. When done, click on the edit button to exit editing mode. Remember to save your project to save the Palette!

2017–09–06 Page published with limited editorial review

Painting Tilemaps

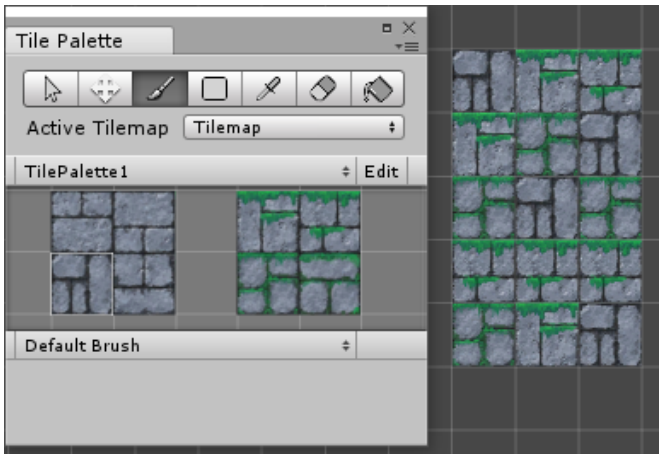
Leave feedback

To paint on a Tilemap it must be the selected Active Tilemap in the Tile Palette. Tilemaps in the **Scene** are automatically added to the list.



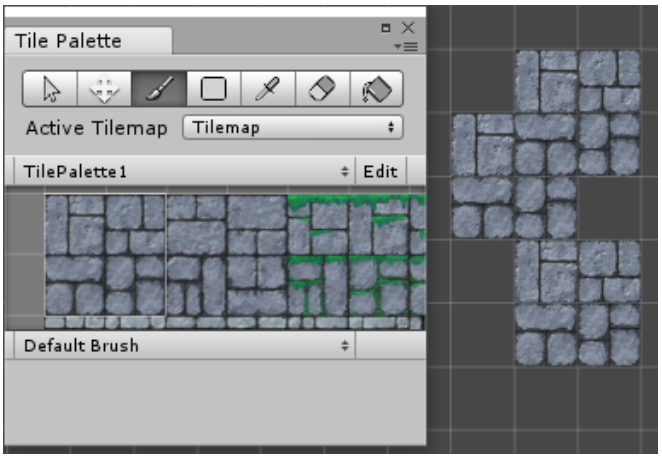
The Tile painting tools are found on the Tilemap Palette

Click on the **Paint Tool** icon, select a Tile from the Tilemap Palette and Left-click on the Tilemap in the **Scene View** to start laying out Tiles.

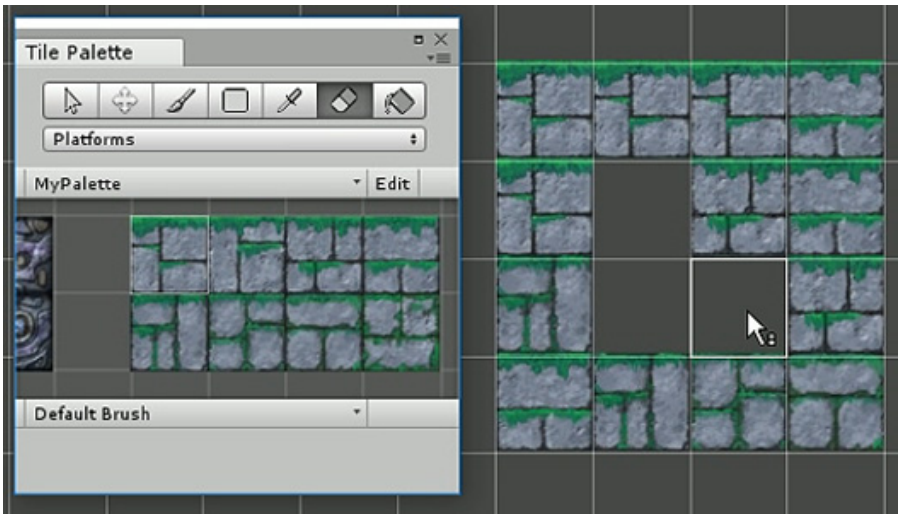


The **Paint Tool**

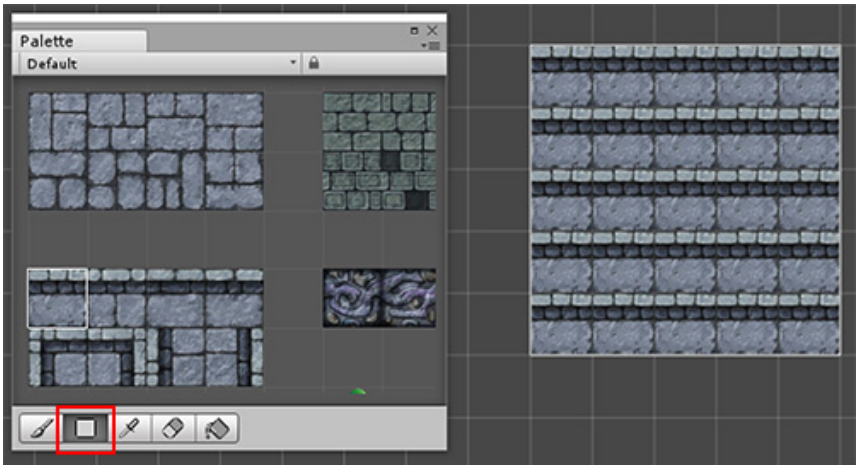
A selection of Tiles can be painted with the Paint Tool. Left-click and drag in the Tilemap Palette to make a selection.



Holding Shift while using the Paint Tool toggles the Erase Tool.

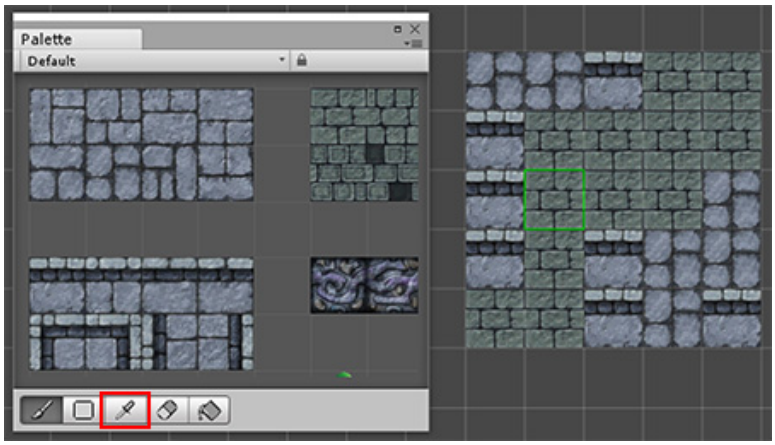


The **Rectangle Tool** draws a rectangular shape on the Tilemap and fills it with the selected tiles.



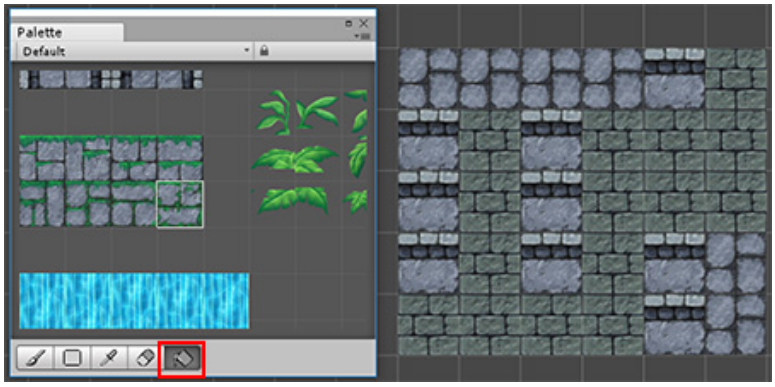
The **Rectangle Tool**

The **Picker Tool** is used to pick Tiles from the Tilemap to paint with. Left-click and drag to select multiple Tiles. Hold the **Ctrl** key (or **Cmd** on macOS) while in **Paint Tool** mode to toggle the **Picker Tool**.

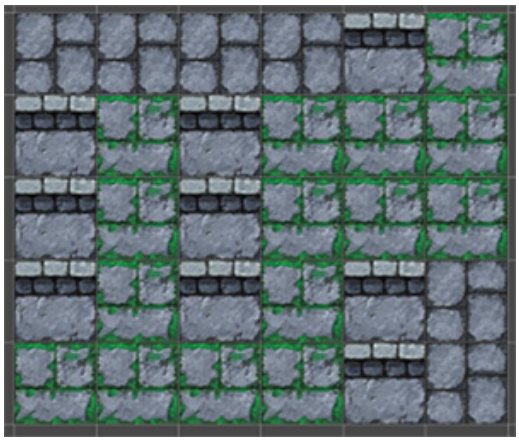


The **Picker Tool**

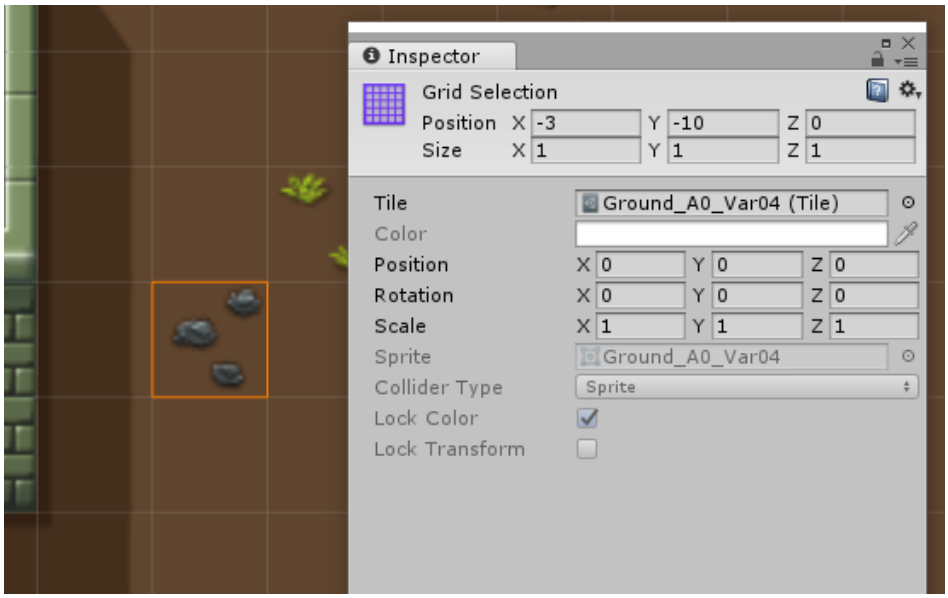
The **Fill Tool** is used to fill an area of Tiles with the selected Tile.



The **Fill Tool**

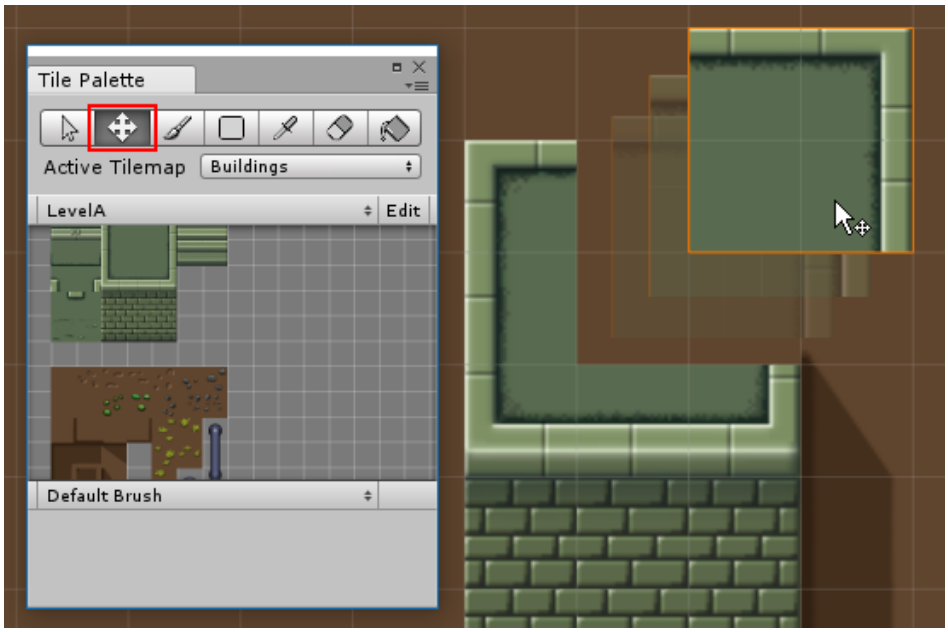


The **Select Tool** is used to select an area of Tiles to be inspected.



The **Select Tool**

The **Move Tool** is used to move a selected area of tiles to another position. Click and drag the selected area to move the Tiles.



The **Move Tool**

Tilemap Focus mode

If you have many Tilemap layers but would like to work solely on a specific layer to work on, you can focus on that particular layer and block out all other **GameObject** from view.

Select the target Tilemap GameObject from the Active Target dropdown in the Palette window or from the Hierarchy window. In the bottom right corner of the SceneView, there is a Tilemap overlay window.

Change the **Focus On** target in the dropdown:

None - No GameObject is focused.
Tilemap - The target Tilemap GameObject is focused. All other GameObjects are faded away. This is good if you want to focus on a single Tilemap layer.
Grid - The parent Grid GameObject with all its children is focused. All other GameObjects are faded away. This is good if you want to focus on the entire Tilemap as a whole.

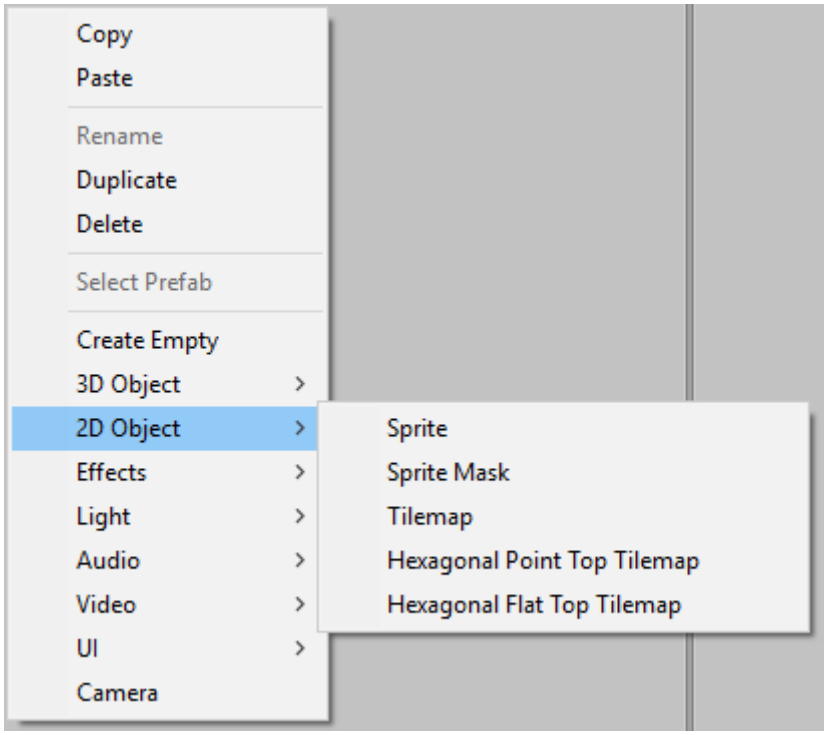
2017–09–06 Page published with limited editorial review

Hexagonal Tilemaps

Leave feedback

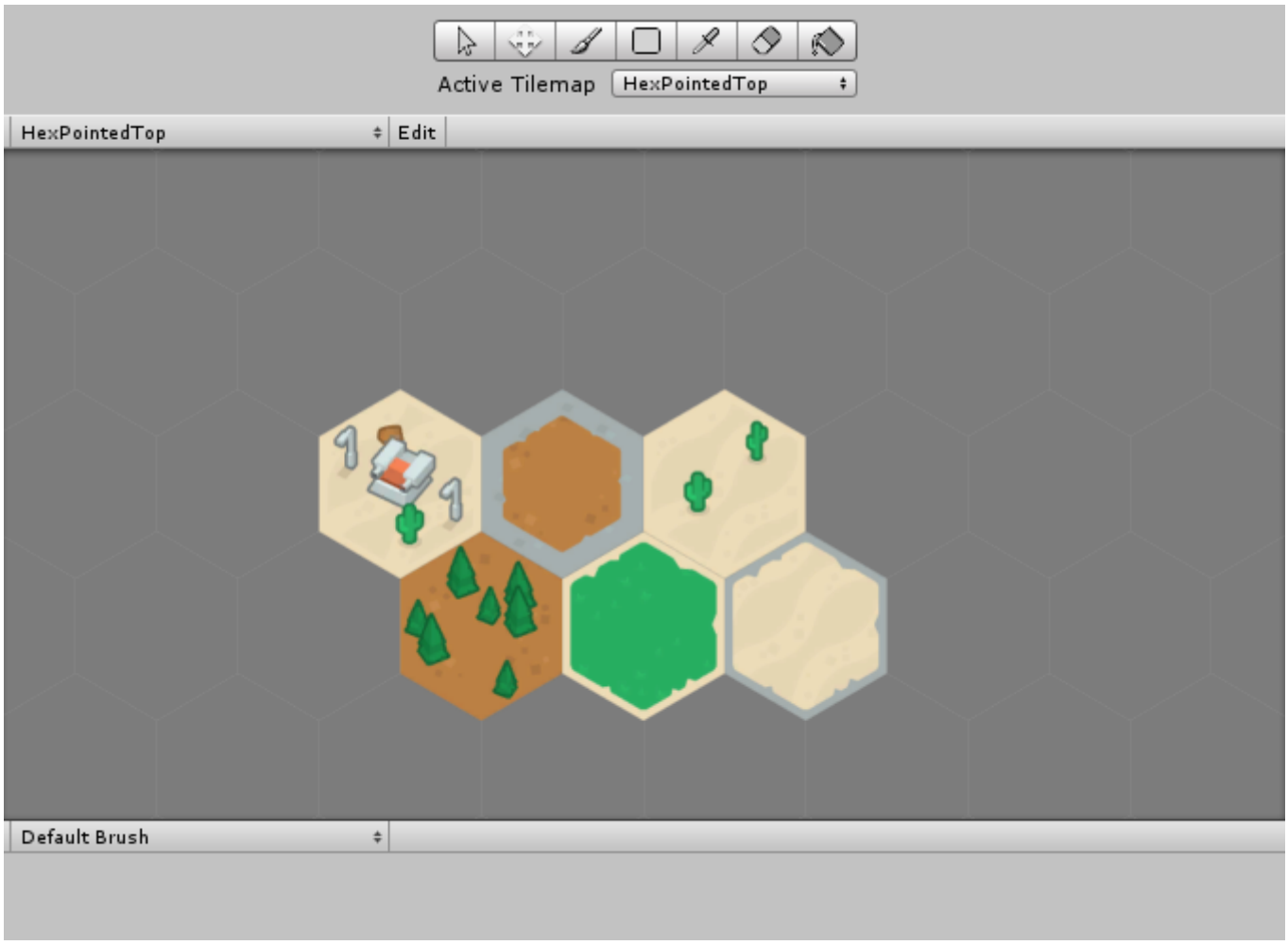
In addition to regular Tilemaps, Unity provides both **Hexagonal Point Top** and **Hexagonal Flat Top** Tilemaps. Hexagonal tiles are often used in strategic tabletop games, because they have consistent distance between their centres and any point on their edge, and neighboring tiles always share edges. This makes them ideal for constructing almost any kind of large play area and allows players to make tactical decisions regarding movement and positioning.

To create a **Hexagonal Tilemap**, follow the same steps to create a regular Tilemap (menu: **GameObject > 2D Object**) but choose one of the **Hexagonal** options in the **2D Object** menu.

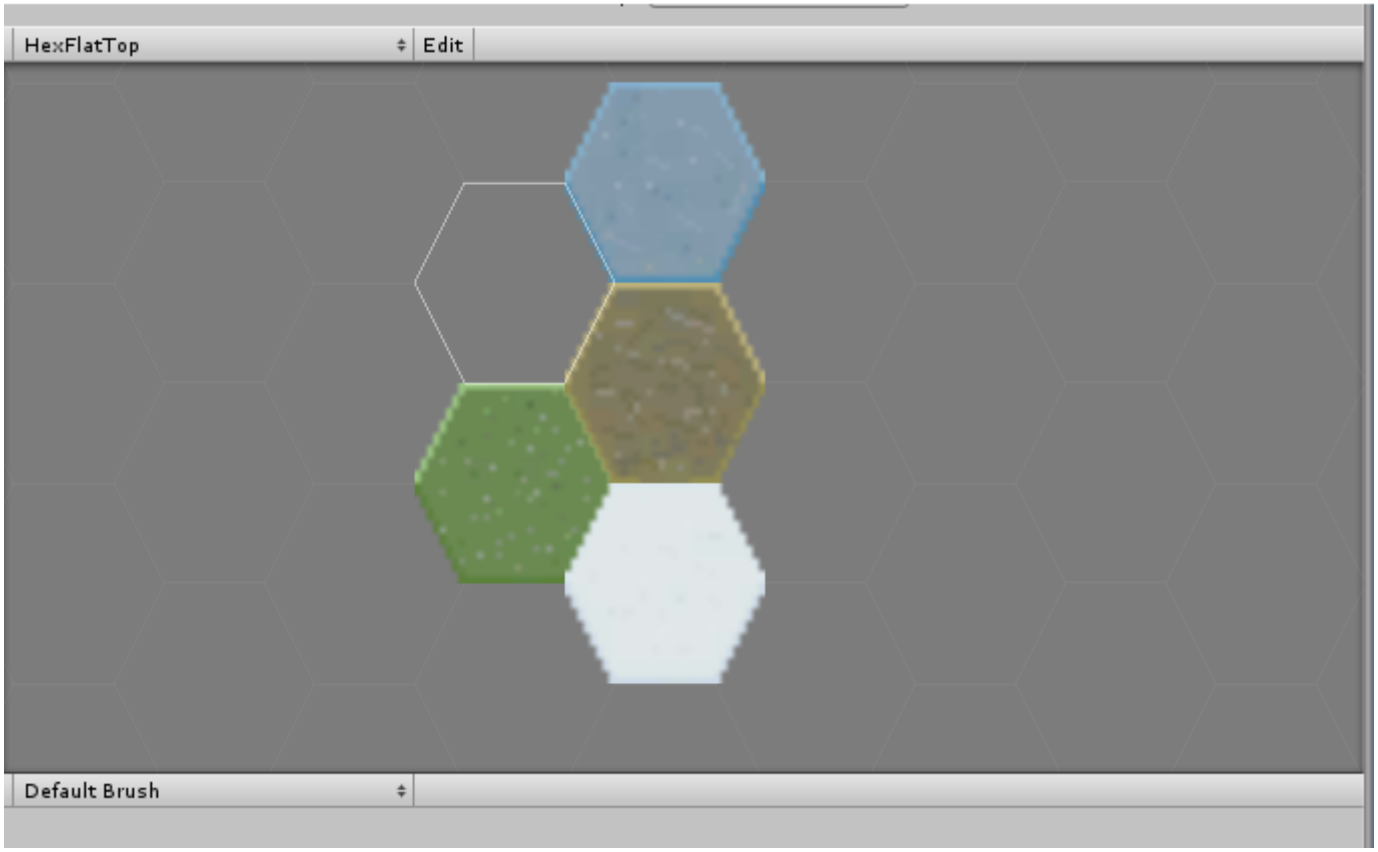


Hexagonal Tilemap options in the 2D Object menu

Select the **Hexagonal Tilemap** option that matches the orientation of the hexagonal Tiles you are using. The following are examples of a **Hexagonal Point Top Tilemap** and a **Hexagonal Flat Top Tilemap**.

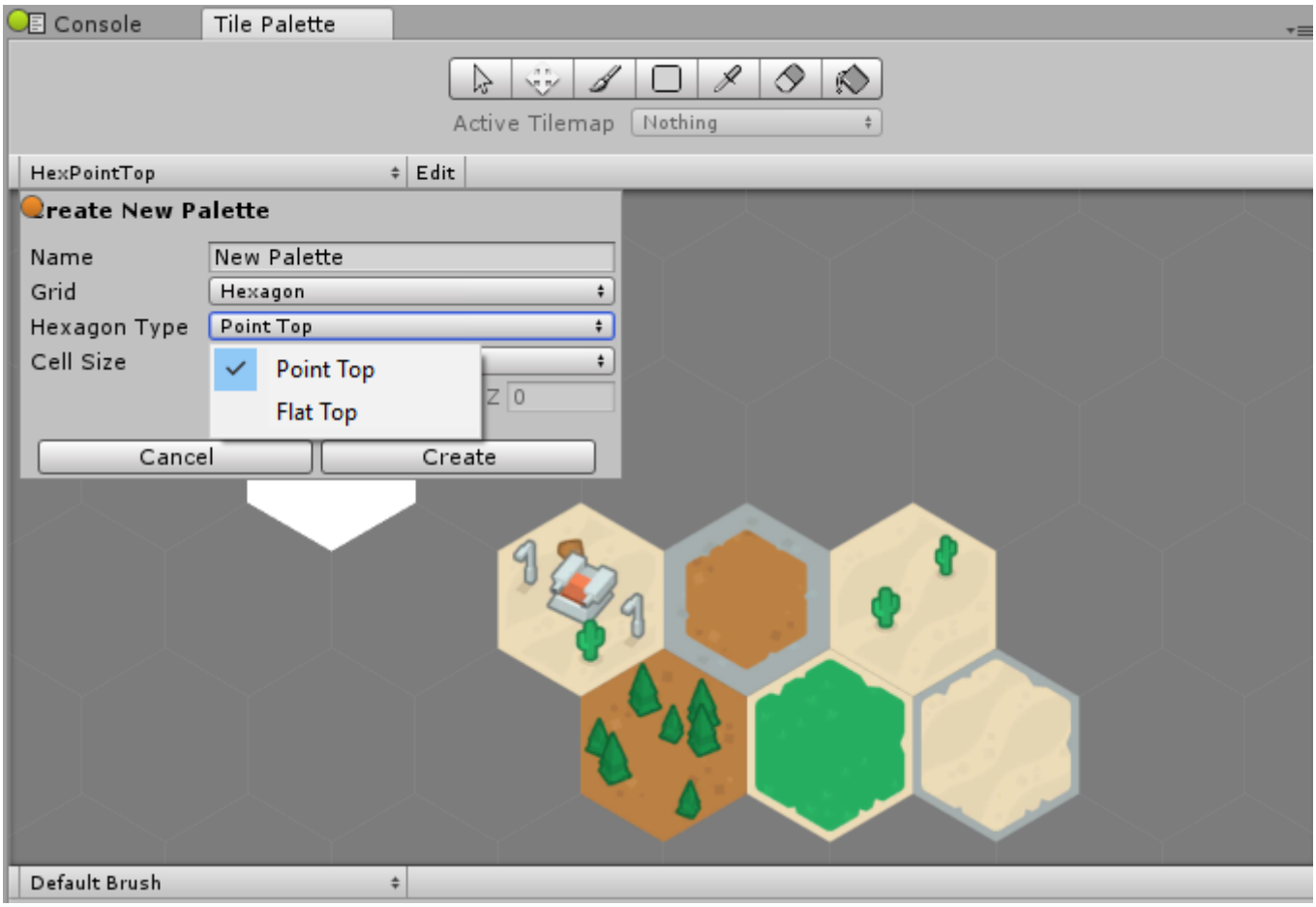


Example of hexagonal Tiles oriented with points facing top



Example of hexagonal Tiles oriented with flat sides facing top

When creating the **Tile Palette** for a Hexagonal Tilemap, set the **Grid** setting of the **Tile Palette** to **Hexagon** and select the **Hexagon Type** that matches the Tilemap and Tiles you are using, as shown below.



Hexagon Type must match the orientation of the hexagonal Tiles

---

2018–07–17 Page published with editorial review

Hexagonal Tilemaps added in 2018.2

Tilemaps and Physics 2D

Leave feedback

You can add a **Tilemap Collider** 2D **component** to the **GameObject** of a **Tilemap** to generate a **collider** based on the Tiles of the **Tilemap**.

A Tilemap Collider 2D component functions like a normal Collider 2D component. You can add Effector 2Ds to modify the behavior of the Tilemap Collider 2D. You can also composite the Tilemap Collider 2D with a Composite Collider 2D.

Adding or removing Tiles with **Collider Type** set to **Sprite** or **Grid** adds or removes the Tile's collider shape from the Tilemap Collider 2D component on the next `LateUpdate` for the Tilemap Collider 2D. This also happens when the Tilemap Collider 2D is being composited by a Composite Collider 2D.

2017–09–06 Page published with limited editorial review

Scriptable Tiles

Leave feedback

Creating Scriptable Tiles

Create a new class inheriting from `TileBase` (or any useful sub-classes of `TileBase`, like `Tile`). Override any required methods for your new `Tile` class. The following are the usual methods you would override:

`RefreshTile` determines which Tiles in the vicinity are updated when this Tile is added to the **Tilemap**.
`GetTileData` determines what the Tile looks like on the Tilemap.
Create instances of your new class using `ScriptableObject.CreateInstance<YOUR_TILE_CLASS>()`. You may convert this new instance to an Asset in the Editor in order to use it repeatedly by calling `AssetDatabase.CreateAsset()`.

You can also make a custom editor for your Tile. This works the same way as custom editors for scriptable objects.

Remember to save your project to ensure that your new Tile **Assets** are saved!

2017–09–06 Page published with limited editorial review

TileBase

Leave feedback

All tiles to be added to the **Tilemap** must inherit from `TileBase`. `TileBase` provides a fixed set of APIs to the Tilemap to communicate its **rendering** properties. For most cases of the APIs, the location of the Tile and the instance of the Tilemap the Tile is placed on is passed in as arguments of the API. You may use this to determine any required attributes for setting the tile information.

```
public void RefreshTile(Vector3Int location, ITilemap tilemap)
```

`RefreshTile` determines which Tiles in the vicinity are updated as this Tile is added to the Tilemap. By default, the `TileBase` calls `tilemap.RefreshTile(location)` to refresh the tile at the current location. Override this to determine which Tiles need to be refreshed due to the placement of the new Tile.

Example: There is a straight road, and you place a `RoadTile` next to it. The straight road isn't valid anymore. It needs a T-section instead. Unity doesn't automatically know what needs to be refreshed, so `RoadTile` needs to trigger the refresh onto itself, but also onto the neighboring road.

```
public bool GetTileData(Vector3Int location, ITilemap tilemap, ref TileData tileData)
```

`GetTileData` determines what the Tile looks like on the Tilemap. See `TileData` below for more details.

```
public bool GetTileAnimationData(Vector3Int location, ITilemap tilemap, ref TileAnimationDa
```

`GetTileAnimationData` determines whether or not the Tile is animated. Return true if there is an animation for the Tile, other returns false if there is not.

```
public bool StartUp(Vector3Int location, ITilemap tilemap, GameObject go)
```

`StartUp` is called for each tile when the Tilemap updates for the first time. You can run any start up logic for Tiles on the Tilemap if necessary. The argument *go* is the instanced version of the object passed in as **gameobject** when `GetTileData` was called. You may update *go* as necessary as well.

2017–09–06 Page published with limited editorial review

Tile

Leave feedback

The `Tile` class is a simple class that allows a sprite to be rendered on the **Tilemap**. Tile inherits from `TileBase`. The following is a description of the methods that are overridden to have the Tile's behaviour.

```
public Sprite sprite;
public Color color = Color.white;
public Matrix4x4 transform = Matrix4x4.identity;
public GameObject gameobject = null;
public TileFlags flags = TileFlags.LockColor;
```

```
public ColliderType colliderType = ColliderType.Sprite;
```

These are the default properties of a Tile. If the Tile was created by dragging and dropping a Sprite onto the Tilemap Palette, the Tile would have the **Sprite** property set as the sprite that was dropped in. You may adjust the properties of the Tile instance to get the Tile required.

```
public void RefreshTile(Vector3Int location, ITilemap tilemap)
```

This is not overridden from `TileBase`. By default, it only refreshes the Tile at that location.

```
public override void GetTileData(Vector3Int location, ITilemap tilemap, ref Tile
{
    tileData.sprite = this.sprite;
    tileData.color = this.color;
    tileData.transform = this.transform;
    tileData.gameobject = this.gameobject;
    tileData.flags = this.flags;

tileData.colliderType = this.colliderType;
}
```

This fills in the required information for Tilemap to render the Tile by copying the properties of the Tile instance into `tileData`.

```
public bool GetTileAnimationData(Vector3Int location, ITilemap tilemap, ref Tile
```

This is not overridden from TileBase. By default, the Tile class does not run any Tile animation and returns false.

```
public bool StartUp(Vector3Int location, ITilemap tilemap, GameObject go)
```

This is not overridden from `TileBase`. By default, the `Tile` class does not have any special start up functionality. If `tileData.gameobject` is set, the Tilemap still instantiates it on start up and place it at the location of the Tile.

2017–09–06 Page published with limited editorial review

TileData

Leave feedback

```
public Sprite sprite
```

This is the **sprite** that is rendered for the Tile.

```
public Color color
```

This is the color that tints the sprite used for the Tile.

```
public Matrix4x4 transform
```

This is the transform matrix used to determine the final location of the Tile. Modify this to add rotations or scaling to the tile.

```
public GameObject gameobject
```

This is the **GameObject** that is instanced when the Tile is added to the **Tilemap**.

```
public TileFlags flags
```

These are the flags which controls the Tile's behaviour. See TileFlags above for more details.

```
public Tile.ColliderType colliderType
```

This controls the **collider** shape generated by the Tile for an attached Tilemap **Collider** 2D **component**. See documentation on `Tile.ColliderType` for more details.

2017–09–06 Page published with limited editorial review

TileAnimationData

Leave feedback

```
public Sprite[] animatedSprites
```

This is an array of **sprites** for the Tile animation. The Tile is animated by these **sprites** in sequential order.

```
public float animationSpeed
```

This is the speed where the Tile animation is run. This is combined with the **Tilemap**'s animation speed for the actual speed.

```
public float animationTimeOffset
```

This allows you to start the animation at a different time frame.

2017–09–06 Page published with limited editorial review

Other useful classes

Leave feedback

TileFlags

`None = 0`

No flags are set for the Tile. This is the default for most Tiles.

`LockColor = 1 << 0`

Set this flag if the Tile script controls the color of the Tile. If this is set, the Tile controls the color as it is placed onto the **Tilemap**. You cannot change the Tile's color through painting or using **scripts**. If this is not set, you can change the Tile's color through painting or using **scripts**.

`LockTransform = 1 << 1`

Set this flag if the Tile script controls the transform of the Tile. If this is set, the Tile controls the transform as it is placed onto the Tilemap. You cannot rotate or change the Tile's transform through painting or using scripts. If this is not set, you can change the Tile's transform through painting or using scripts.

`LockAll = LockColor | LockTransform`

This is a combination of all the lock flags used by TileBase.

`InstantiateSpawnGameObjectRuntimeOnly = 1 << 2`

Set this flag if the Tile script should spawn its game object only when your project is running and not in Editor mode.

Tile.ColliderType

`None = 0`

This Tile does not generate a **collider** shape.

`Sprite = 1`

The collider shape generated by this Tile is the physics shape set by the **Sprite** that the Tile returns. If no physics shape has been set in the **Sprite**, it tries to generate a shape based on the outline of the **Sprite**.

Note: If a collider shape for a Tile needs to be generated in runtime, set a physics shape for the Sprite or set the Texture of the Sprite to be readable in order for Unity to generate a shape based on the outline.

`Grid = 2`

The collider shape generated by this Tile is the shape of the cell, defined by the layout of the Grid.

ITilemap

`ITilemap` is the base class where `Tile` can retrieve data from the Tilemap when the Tilemap tries to retrieve data from the Tile.

```
Vector3Int origin { get; }
```

This returns the origin point of the Tilemap in cellspace.

```
Vector3Int size { get; }
```

This returns the size of the Tilemap in cellspace.

```
Bounds localBounds { get; }
```

This returns the bounds of the Tilemap in localspace.

```
BoundsInt cellBounds { get; }
```

This returns the bounds of the Tilemap in cellspace.

```
Sprite GetSprite(Vector3Int location);
```

This returns the sprite used by the Tile in the Tilemap at the given location.

```
Color GetColor(Vector3Int location);
```

This returns the color used by the Tile in the Tilemap at the given location.

```
Matrix4x4 GetTransformMatrix(Vector3Int location);
```

This returns the transform matrix used by the Tile in the Tilemap at the given location.

```
TileFlags GetTileFlags(Vector3Int location);
```

This returns the tile flags used by the Tile in the Tilemap at the given location.

```
TileBase GetTile(Vector3Int location);
```

This returns the Tile in the Tilemap at the given location. If there is no Tile there, it returns null.

```
T GetTile<T>(Vector3Int location) where T : TileBase;
```

This returns the Tile in the Tilemap at the given location with type T. If there is no Tile with the matching type there, it returns null.

```
void RefreshTile(Vector3Int location);
```

This requests a refresh of the Tile in the Tilemap at the given location.

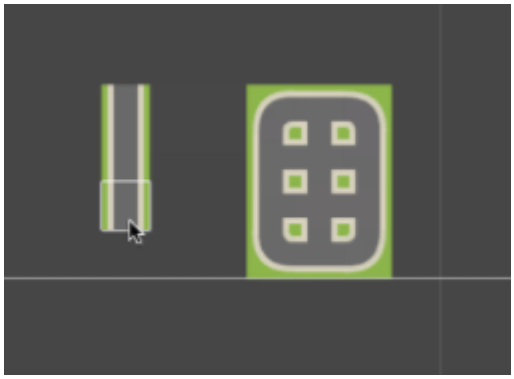
```
T GetComponent<T>();
```

This returns the component T that is attached to the GameObject of the Tilemap.

2017–09–06 Page published with limited editorial review

Scriptable Tile example

Leave feedback



The `RoadTile` example provide the ability to easily layout linear segments onto the **Tilemap**, such as roads or pipes, with a minimal set of **sprites**. The following is a script used to create the Tile.

```
using UnityEngine;
using System.Collections;
#if UNITY_EDITOR
using UnityEditor;
#endif

public class RoadTile : Tile
{
    public Sprite[] m_Sprites;
    public Sprite m_Preview;
    // This refreshes itself and other RoadTiles that are orthogonally and diagor
    public override void RefreshTile(Vector3Int location, ITilemap tilemap)
    {
        for (int yd = -1; yd <= 1; yd++)
            for (int xd = -1; xd <= 1; xd++)
            {
                Vector3Int position = new Vector3Int(location.x + xd, location.y
                if (HasRoadTile(tilemap, position))
                    tilemap.RefreshTile(position);
            }
    }
    // This determines which sprite is used based on the RoadTiles that are adjac
    // As the rotation is determined by the RoadTile, the TileFlags.OverrideTrans
    public override void GetTileData(Vector3Int location, ITilemap tilemap, ref T
    {
        int mask = HasRoadTile(tilemap, location + new Vector3Int(0, 1, 0)) ? 1 :
        mask += HasRoadTile(tilemap, location + new Vector3Int(1, 0, 0)) ? 2 : 0;
        mask += HasRoadTile(tilemap, location + new Vector3Int(0, -1, 0)) ? 4 : 0
        mask += HasRoadTile(tilemap, location + new Vector3Int(-1, 0, 0)) ? 8 : 0
        int index = GetIndex((byte)mask);
        if (index >= 0 && index < m_Sprites.Length)
        {
```

```
            tileData.sprite = m_Sprites[index];
            tileData.color = Color.white;
            var m = tileData.transform;
            m.SetTRS(Vector3.zero, GetRotation((byte) mask), Vector3.one);
            tileData.transform = m;
            tileData.flags = TileFlags.LockTransform;
            tileData.colliderType = ColliderType.None;
        }
        else
        {
        Debug.LogWarning("Not enough sprites in RoadTile instance");
}
    }
    // This determines if the Tile at the position is the same RoadTile.
    private bool HasRoadTile(ITilemap tilemap, Vector3Int position)
    {
        return tilemap.GetTile(position) == this;
    }
    // The following determines which sprite to use based on the number of adjace
    private int GetIndex(byte mask)
    {
        switch (mask)
        {
            case 0: return 0;
            case 3:
            case 6:
            case 9:
            case 12: return 1;
            case 1:
            case 2:
            case 4:
            case 5:
            case 10:
            case 8: return 2;
            case 7:
            case 11:
            case 13:
            case 14: return 3;
            case 15: return 4;
        }
        return -1;
    }
// The following determines which rotation to use based on the positions of adjac
    private Quaternion GetRotation(byte mask)
    {
        switch (mask)
        {
```

```
            case 9:
            case 10:
            case 7:
            case 2:
            case 8:
                return Quaternion.Euler(0f, 0f, -90f);
            case 3:
            case 14:
                return Quaternion.Euler(0f, 0f, -180f);
            case 6:
            case 13:
                return Quaternion.Euler(0f, 0f, -270f);
        }
        return Quaternion.Euler(0f, 0f, 0f);
    }
#if UNITY_EDITOR
// The following is a helper that adds a menu item to create a RoadTile Asset
    [MenuItem("Assets/Create/RoadTile")]
    public static void CreateRoadTile()
    {
        string path = EditorUtility.SaveFilePanelInProject("Save Road Tile", "New
        if (path == "")
            return;
    AssetDatabase.CreateAsset(ScriptableObject.CreateInstance<RoadTile>(), path);
    }
#endif
}
```

2017–09–06 Page published with limited editorial review

Scriptable Brushes

Leave feedback

Creating Scriptable Brushes

Create a new class inheriting from `GridBrushBase` (or any useful sub-classes of `GridBrushBase` like `GridBrush`). Override any required methods for your new `Brush` class. The following are the usual methods you would override:

`Paint` allows the Brush to add items onto the target Grid
`Erase` allows the Brush to remove items from the target Grid
`FloodFill` allows the Brush to fill items onto the target Grid
`Rotate` rotates the items set in the Brush.
`Flip` flips the items set in the Brush.
Create instances of your new class using `ScriptableObject.CreateInstance<(Your Brush Class>()`. You may convert this new instance to an Asset in the Editor in order to use it repeatedly by calling `AssetDatabase.CreateAsset()`.

You can also make a custom editor for your brush. This works the same way as custom editors for scriptable objects. The following are the main methods you would want to override when creating a custom editor:

You can override `OnPaintInspectorGUI` to have an **inspector** show up on the Palette when the Brush is selected to provide additional behaviour when painting.
You can also override `OnPaintSceneGUI` to add additional behaviour when painting on the `SceneView`.
You can also override `validTargets` to have a custom list of targets which the Brush can interact with. This list of targets is shown as a dropdown in the Palette window.
When created, the Scriptable Brush is listed in the **Brushes Dropdown** in the Palette window. By default, an instance of the Scriptable Brush script is instantiated and stored in the *Library* folder of your project. Any modifications to the brush properties are stored in that instance. If you want to have multiple copies of that Brush with different properties, you can instantiate the Brush as **Assets** in your project. These Brush **Assets** are listed separately in the Brush dropdown.

You can add a `CustomGridBrush` attribute to your Scriptable Brush class. This allows you to configure the behaviour of the Brush in the Palette window. The `CustomGridBrush` attribute has the following properties:

`HideAssetInstances` - Setting this to true hides all copies of created Brush Assets in the Palette window. Set this if you only want the default instance to show up in the Brush dropdown in the Palette window.
`HideDefaultInstances` - Setting this to true hides the default instance of the Brush in the Palette window. Set this if you only want created Assets to show up in the Brush dropdown in the Palette window.
`DefaultBrush` - Setting this to true sets the default instance of the Brush as the default Brush in the project. This makes this Brush the default selected Brush whenever the project starts up. Only set one Scriptable Brush to be the Default Brush!
`DefaultName` - Setting this makes the Brush dropdown use this as the name for the Brush instead of the name of the class of the Brush.

Remember to save your project to ensure that your new Brush Assets are saved!

2017–09–06 Page published with limited editorial review

GridBrushBase

Leave feedback

All brushes added must inherit from `GridBrushBase`. `GridBrushBase` provides a fixed set of APIs for painting.

```
public virtual void Paint(GridLayout grid, GameObject brushTarget, Vector3Int po
```

`Paint` adds data onto the target GameObject `brushTarget` with the `GridLayout` grid at the given position. This is triggered when the Brush is activated on the grid and the **Paint Tool** is selected on the Palette window. Override this to implement the action desired on painting.

```
public virtual void Erase(GridLayout grid, GameObject brushTarget, Vector3Int po
```

`Erase` removes data onto the target **GameObject** `brushTarget` with the `GridLayout` grid at the given position. This is triggered when the Brush is activated on the grid and the **Erase Tool** is selected on the Palette window. Override this to implement the action desired on erasing.

```
public virtual void BoxFill(GridLayout grid, GameObject brushTarget, BoundsInt p
```

`BoxFill` adds data onto the target GameObject `brushTarget` with the `GridLayout` grid onto the given bounds. This is triggered when the Brush is activated on the grid and the **Box Fill Tool** is selected on the Palette window. Override this to implement the action desired on filling.

```
public virtual void FloodFill(GridLayout grid, GameObject brushTarget, Vector3In
```

`FloodFill` adds data onto the target GameObject `brushTarget` with the `GridLayout` grid starting at the given position and filling all other possible areas linked to the position. This is triggered when the Brush is activated on the grid and the **Flood Fill Tool** is selected on the Palette window. Override this to implement the action desired on filling.

```
public virtual void Rotate(RotationDirection direction)
```

`Rotate` rotates the content in the brush with the given direction based on the currently set pivot.

```
public virtual void Flip(FlipAxis flip)
```

`Flip` flips the content of the brush with the given axis based on the currently set pivot.

```
public virtual void Select(GridLayout grid, GameObject brushTarget, BoundsInt po
```

`Select` marks a boundary on the target GameObject `brushTarget` with the `GridLayout` grid from the given bounds. This allows you to view information based on the selected boundary and move the selection with the **Move Tool**. This is triggered when the Brush is activated on the grid and the Select tool is selected on the Palette window. Override this to implement the action desired when selecting from a target.

```
public virtual void Pick(GridLayout grid, GameObject brushTarget, BoundsInt posi
```

`Pick` pulls data from the target GameObject `brushTarget` with the `GridLayout` grid from the given bounds and pivot position, and fills the brush with that data. This is triggered when the Brush is activated on the grid and the **Pick Tool** is selected on the Palette window. Override this to implement the action desired when picking from a target.

```
public virtual void Move(GridLayout grid, GameObject brushTarget, BoundsInt from
```

`Move` marks the movement from the target GameObject `brushTarget` with the `GridLayout` grid from the given starting position to the given ending position. Override this to implement the action desired when moving from a target. This is triggered when the Brush is activated on the grid and the **Move Tool** is selected on the Palette window and the Move is performed (`MouseDrag`). Generally, this would be any behaviour while a `Move` operation from the brush is being performed.

```
public virtual void MoveStart(GridLayout grid, GameObject brushTarget, BoundsInt
```

`MoveStart` marks the start of a move from the target GameObject `brushTarget` with the `GridLayout` grid from the given bounds. This is triggered when the Brush is activated on the grid and the **Move Tool** is selected on the Palette window and the `Move` is first triggered (`MouseDown`). Override this to implement the action desired when starting a move from a target. Generally, this would be picking of data from the target with the given start position.

```
public virtual void MoveEnd(GridLayout grid, GameObject brushTarget, BoundsInt p
```

`MoveEnd` marks the end of a move from the target GameObject `brushTarget` with the `GridLayout` grid from the given bounds. This is triggered when the Brush is activated on the grid and the **Move Tool** is selected on the Palette window and the `Move` is completed (`MouseUp`). Override this to implement the action desired when ending a move from a target. Generally, this would be painting of data to the target with the given end position.

2017–09–06 Page published with limited editorial review

GridBrushEditorBase

Leave feedback

All brush editors added must inherit from `GridBrushEditorBase`. `GridBrushEditorBase` provides a fixed set of APIs for drawing of **inspectors** on the Palette window and drawing of **Gizmos** on the **Scene View**.

```
public virtual GameObject[] validTargets
```

This returns a list of **GameObjects** which are valid targets to be painted on by the brush. This appears in the dropdown in the Palette window. Override this to have a custom list of targets which this Brush can interact with.

```
public virtual void OnPaintInspectorGUI()
```

This displays an inspector for editing Brush options in the Palette. Use this to update brush functionality while editing in the **Scene** view.

```
public virtual void OnSelectionInspectorGUI()
```

This displays an inspector for when cells are selected on a target Grid. Override this to show a custom inspector view for the selected cells.

```
public virtual void OnPaintSceneGUI(GridLayout grid, GameObject brushTarget, BoundsI
```

This is used for drawing additional gizmos on the Scene View when painting with the brush. Tool is the currently selected tool in the Palette. Executing returns whether the brush is being used at the particular time.

2017–09–06 Page published with limited editorial review

Other useful classes

Leave feedback

GridBrushBase.Tool

`Select = 0` - Tool for Selection for a GridBrush

`Move = 1`- Tool for Moving for a GridBrush

`Paint = 2` - Tool for Painting for a GridBrush

`Box = 3` - Tool for Boxfill for a GridBrush

`Pick = 4` - Tool for Picking for a GridBrush

`Erase = 5` - Tool for Erasing for a GridBrush

`Floodfill = 6` - Tool for Floodfill for a GridBrush

GridBrushBase.RotationDirection

`Clockwise = 0` - Clockwise rotation direction. Use this when rotating a Brush.

`CounterClockwise = 1` - Counter clockwise rotation direction. Use this when rotating a Brush.

GridBrushBase.FlipAxis

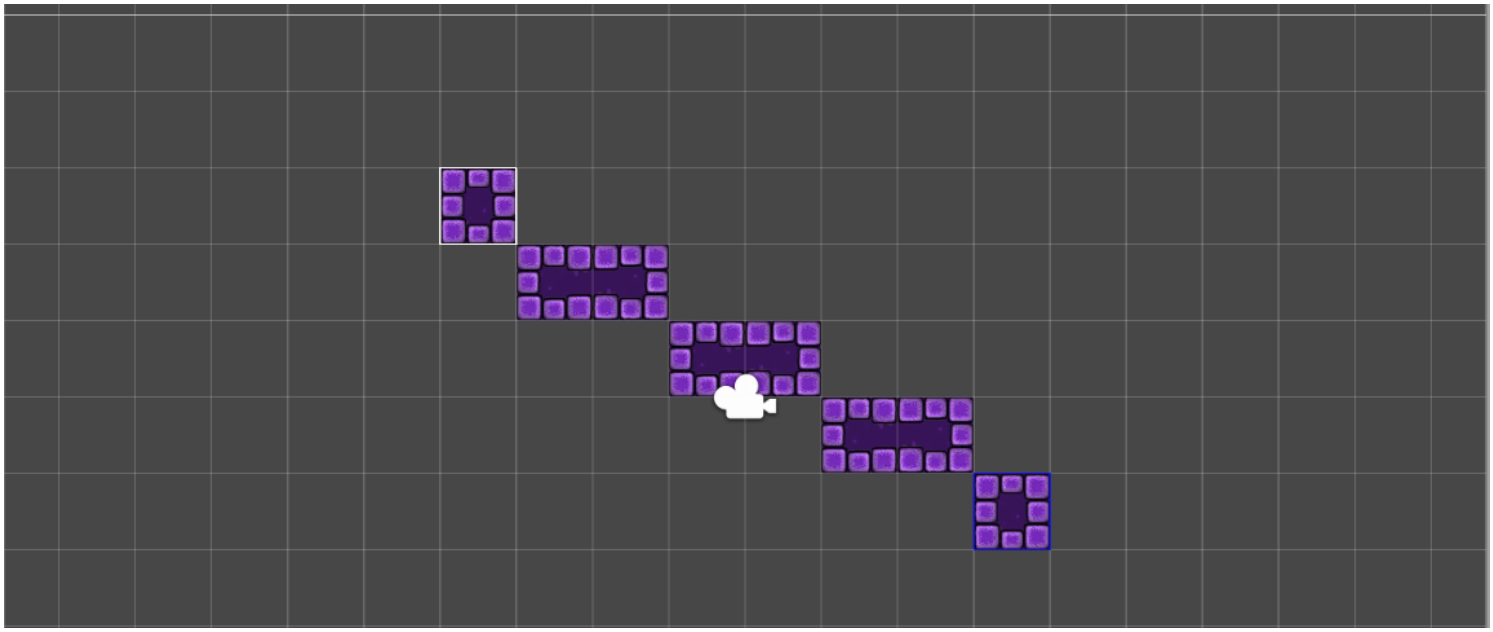
`X = 0` - Flips along the X Axis. Use this when flipping a Brush.

`Y = 1` - Flips along the Y Axis. Use this when flipping a Brush.

2017–09–06 Page published with limited editorial review

Scriptable Brush example

Leave feedback



`LineBrush` provides the ability to easily draw lines of Tiles onto the Tilemap by specifying the start point and the end point. The Paint method for the LineBrush is overridden to allow the user to specify the start of a line with the first mouse click in Paint mode and draw the line with the second mouse click in Paint mode. The `OnPaintSceneGUI` method is overridden to produce the preview of the line to be drawn between the first and second mouse clicks. The following is a script used to create the Brush.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.__Tilemaps__;

namespace UnityEditor
{
    [CustomGridBrush(true, false, false, "Line Brush")]
    public class LineBrush : GridBrush {
        public bool lineStartActive = false;
        public Vector3Int lineStart = Vector3Int.zero;
        public override void Paint(GridLayout grid, GameObject brushTarget, Vect
        {
            if (lineStartActive)
            {
                Vector2Int startPos = new Vector2Int(lineStart.x, lineStart.y);
                Vector2Int endPos = new Vector2Int(position.x, position.y);
                if (startPos == endPos)
                    base.Paint(grid, brushTarget, position);
                else
                {
```

```
                    foreach (var point in GetPointsOnLine(startPos, endPos))
                    {
                        Vector3Int paintPos = new Vector3Int(point.x, point.y, p
                        base.Paint(grid, brushTarget, paintPos);
                    }
                }
                lineStartActive = false;
            }
            else
            {
                lineStart = position;
                lineStartActive = true;
            }
        }
        [MenuItem("Assets/Create/Line Brush")]
        public static void CreateBrush()
        {
            string path = EditorUtility.SaveFilePanelInProject("Save Line Brush"
            if (path == "")
                return;
AssetDatabase.CreateAsset(ScriptableObject.CreateInstance<LineBrush>(), path);
        }
        // http://ericw.ca/notes/bresenhams-line-algorithm-in-csharp.html
        public static IEnumerable<Vector2Int> GetPointsOnLine(Vector2Int p1, Vec
        {
            int x0 = p1.x;
            int y0 = p1.y;
            int x1 = p2.x;
            int y1 = p2.y;
            bool steep = Math.Abs(y1 - y0) > Math.Abs(x1 - x0);
            if (steep)
            {
                int t;
                t = x0; // swap x0 and y0
                x0 = y0;
                y0 = t;
                t = x1; // swap x1 and y1
                x1 = y1;
                y1 = t;
            }
            if (x0 > x1)
            {
                int t;
                t = x0; // swap x0 and x1
                x0 = x1;
                x1 = t;
```

```
            t = y0; // swap y0 and y1
            y0 = y1;
            y1 = t;
        }
        int dx = x1 - x0;
        int dy = Math.Abs(y1 - y0);
        int error = dx / 2;
        int ystep = (y0 < y1) ? 1 : -1;
        int y = y0;
        for (int x = x0; x <= x1; x++)
        {
            yield return new Vector2Int((steep ? y : x), (steep ? x : y));
            error = error - dy;
            if (error < 0)
            {
                y += ystep;
                error += dx;
            }
        }
        yield break;
    }
}
[CustomEditor(typeof(LineBrush))]
public class LineBrushEditor : GridBrushEditor
{
    private LineBrush lineBrush { get { return target as LineBrush; } }
    public override void OnPaintSceneGUI(GridLayout grid, GameObject brushTa
    {
        base.OnPaintSceneGUI(grid, brushTarget, position, tool, executing);
        if (lineBrush.lineStartActive)
        {
            Tilemap tilemap = brushTarget.GetComponent<Tilemap>();
            if (tilemap != null)
                tilemap.ClearAllEditorPreviewTiles();
            // Draw preview tiles for tilemap
            Vector2Int startPos = new Vector2Int(lineBrush.lineStart.x, line
            Vector2Int endPos = new Vector2Int(position.x, position.y);
            if (startPos == endPos)
                PaintPreview(grid, brushTarget, position.min);
            else
            {
                foreach (var point in LineBrush.GetPointsOnLine(startPos, en
                {
                    Vector3Int paintPos = new Vector3Int(point.x, point.y, p
                    PaintPreview(grid, brushTarget, paintPos);
                }
```

```
                }
                if (Event.current.type == EventType.Repaint)
                {
                    var min = lineBrush.lineStart;
                    var max = lineBrush.lineStart + position.size;
                    // Draws a box on the picked starting position
                    GL.PushMatrix();
                    GL.MultMatrix(GUI.matrix);
                    GL.Begin(GL.LINES);
                    Handles.color = Color.blue;
                    Handles.DrawLine(new Vector3(min.x, min.y, min.z), new Vecto
                    Handles.DrawLine(new Vector3(max.x, min.y, min.z), new Vecto
                    Handles.DrawLine(new Vector3(max.x, max.y, min.z), new Vecto
                    Handles.DrawLine(new Vector3(min.x, max.y, min.z), new Vecto
                    GL.End();
                    GL.PopMatrix();
                }
            }
        }
    }
}
```

2017–09–06 Page published with limited editorial review

Physics Reference 2D

Leave feedback

This section gives details of the **components** used with 2D physics. For information on the equivalent 3D **components**, see Physics 3D Reference.

To specify 2D physics settings, see Physics 2D Settings.

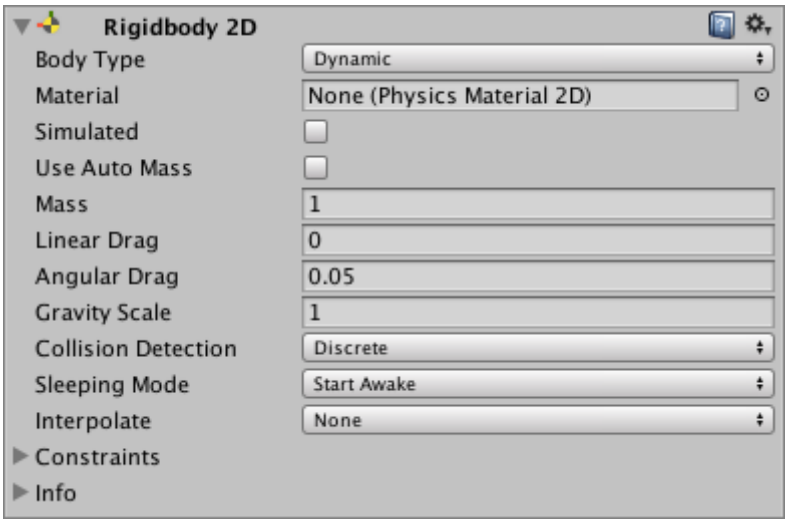
2018–04–24 Page amended with editorial review

Rigidbody 2D

Leave feedback

SWITCH TO SCRIPTING

A Rigidbody 2D **component** places an object under the control of the **physics engine**. Many concepts familiar from the standard Rigidbody component carry over to Rigidbody 2D; the differences are that in 2D, objects can only move in the XY plane and can only rotate on an axis perpendicular to that plane.



The Rigidbody 2D component. This appears differently in the Unity Editor depending on which Body Type you have selected. See Body Type, below, to learn more.

How a Rigidbody 2D works

Usually, the Unity Editor's **Transform component** defines how a **GameObject** (and its child GameObjects) is positioned, rotated and scaled within the **Scene**. When it is changed, it updates other components, which may update things like where they render or where **colliders** are positioned. The 2D physics engine is able to move **colliders** and make them interact with each other, so a method is required for the physics engine to communicate this movement of **colliders** back to the Transform components. This movement and connection with **colliders** is what a Rigidbody 2D component is for.

The Rigidbody 2D component overrides the Transform and updates it to a position/rotation defined by the Rigidbody 2D. Note that while you can still override the Rigidbody 2D by modifying the Transform component yourself (because Unity exposes all properties on all components), doing so will cause problems such as GameObjects passing through or into each other, and unpredictable movement.

Any Collider 2D component added to the same GameObject or child GameObject is implicitly attached to that Rigidbody 2D. When a Collider 2D is attached to the Rigidbody 2D, it moves with it. A Collider 2D should never be moved directly using the Transform or any collider offset; the Rigidbody 2D should be moved instead. This offers the best performance and ensures correct **collision** detection. Collider 2Ds attached to the same Rigidbody 2D won't collide with each other. This means you can create a set of colliders that act effectively as a single compound collider, all moving and rotating in sync with the Rigidbody 2D.

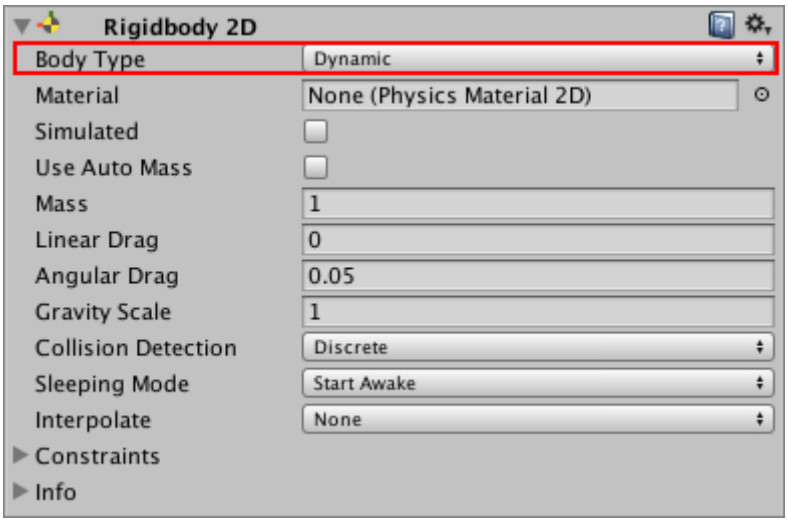
When designing a Scene, you are free to use a default Rigidbody 2D and start attaching colliders. These colliders allow any other colliders attached to different Rigidbody 2Ds to collide with each other.

Tip

Adding a **Rigidbody 2D** allows a **sprite** to move in a physically convincing way by applying forces from the scripting API. When the appropriate collider component is also attached to the **sprite** GameObject, it is affected by collisions with other moving GameObjects. Using physics simplifies many common gameplay mechanics and allows for realistic behavior with minimal coding.

Body Type

The Rigidbody 2D component has a setting at the top labelled **Body Type**. The option you choose for this affects the other settings available on the component.



There are three options for **Body Type**; each defines a common and fixed behavior. Any Collider 2D attached to a Rigidbody 2D inherits the Rigidbody 2D's **Body Type**. The three options are:

Dynamic
Kinematic
Static

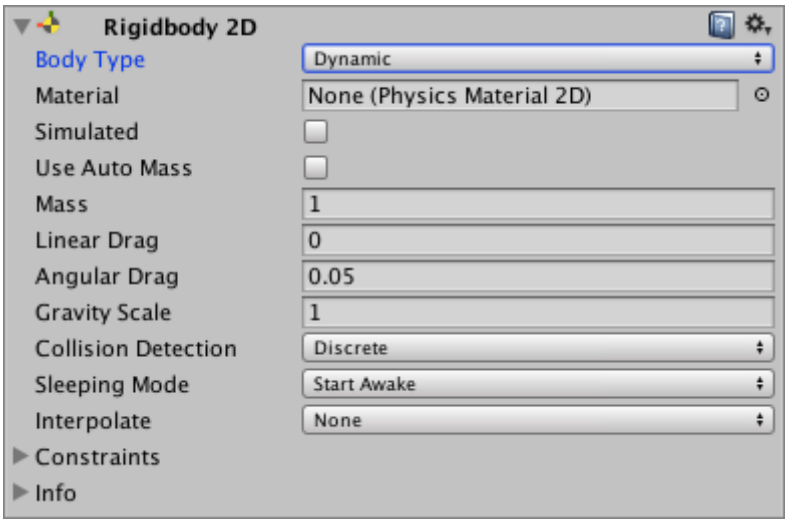
The option you choose defines:

Movement (position & rotation) behavior
Collider interaction

Note that although Rigidbody 2Ds are often described as colliding with each other, it is the Collider 2Ds attached to each of those bodies which collide. Rigidbody 2Ds cannot collide with each other without colliders.

Changing the Body Type of a Rigidbody 2D can be a tricky process. When a Body Type changes, various mass-related internal properties are recalculated immediately, and all existing contacts for the Collider 2Ds attached to the Rigidbody 2D need to be re-evaluated during the GameObject's next FixedUpdate. Depending on how many contacts and Collider 2Ds are attached to the body, changing the Body Type can cause variations in performance.

Body Type: Dynamic



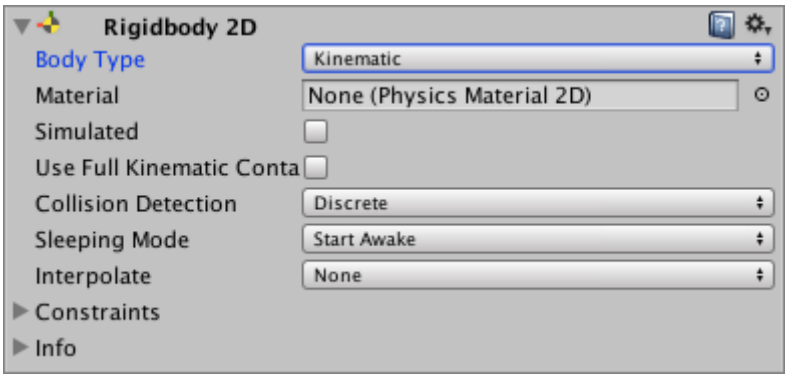
A **Dynamic** Rigidbody 2D is designed to move under simulation. It has the full set of properties available to it such as finite mass and drag, and is affected by gravity and forces. A Dynamic body will collide with every other body type, and is the most interactive of body types. This is the default body type for a Rigidbody 2D, because it is the most common body type for things that need to move. It's also the most performance-expensive body type, because of its dynamic nature and interactivity with everything around it. All Rigidbody 2D properties are available with this body type.

Do not use the Transform component to set the position or rotation of a **Dynamic** Rigidbody 2D. The simulation repositions a **Dynamic** Rigidbody 2D according to its velocity; you can change this directly via forces applied to it by **scripts**, or indirectly via collisions and gravity.

Property:	**Function:**
Body Type	Set the RigidBody 2D's component settings, so that you can manipulate movement (position and rotation) behavior and Collider 2D interaction. Options are: **Dynamic**, **Kinematic**, **Static**
Material	Use this to specify a common material for all Collider 2Ds attached to a specific parent Rigidbody 2D. **Note:** A Collider 2D uses its own Material property if it has one set. If there is no Material specified here or in the Collider 2D, the default option is **None (Physics Material 2D)**. This uses a default Material which you can set in the Physics 2D Settings window. A Collider 2D uses the following order of priority to determine which **Material** setting to use: 1. A Physics Material 2D specified on the Collider 2D itself. 2. A Physics Material 2D specified on the attached Rigidbody 2D. A Physics Material 2D default material specified in the Physics 2D Settings. **TIP:** Use this to ensure that all Collider 2Ds attached to the same **Static** Body Type Rigidbody 2D can all use the same Material.
Simulated	Enable **Simulated** (check the box) if you want the Rigidbody 2D and any attached Collider 2Ds and **Joint** 2Ds to interact with the physics simulation during run time. If this is disabled (the box is unchecked), these components do not interact with the simulation. See Rigidbody 2D properties: Simulated, below, for more details. This box is checked by default.

Property:	Function:
Use Auto Mass	Check the box if you want the Rigidbody 2D to automatically detect the GameObject's mass from its Collider 2D.
Mass	Define the mass of the Rigidbody 2D. This is grayed out if you have selected Use Auto Mass.
Linear Drag	Drag coefficient affecting positional movement.
Angular Drag	Drag coefficient affecting rotational movement.
Gravity Scale	Define the degree to which the GameObject is affected by gravity.
Collision Detection	Define how collisions between Collider 2D are detected.
Discrete	When you set the **Collision Detection** to **Discrete**, GameObjects with Rigidbody 2Ds and Collider 2Ds can overlap or pass through each other during a physics update, if they are moving fast enough. Collision contacts are only generated at the new position.
Continuous	When the **Collision Detection** is set to **Continuous**, GameObjects with Rigidbody 2Ds and Collider 2Ds do not pass through each other during an update. Instead, Unity calculates the first impact point of any of the Collider 2Ds, and moves the GameObject there. Note that this takes more CPU time than **Discrete**.
Sleeping Mode	Define how the GameObject "sleeps" to save processor time when it is at rest.
Never Sleep	Sleeping is disabled (this should be avoided where possible, as it can impact system resources).
Start Awake	GameObject is initially awake.
Start Asleep	GameObject is initially asleep but can be woken by collisions.
Interpolate	Define how the GameObject's movement is interpolated between physics updates (useful when motion tends to be jerky).
None	No movement smoothing is applied.
Interpolate	Movement is smoothed based on the GameObject's positions in previous frames.
Extrapolate	Movement is smoothed based on an estimate of its position in the next frame.
Constraints	Define any restrictions on the Rigidbody 2D's motion.
Freeze Position	Stops the Rigidbody 2D moving in the world X & Y axes selectively.
Freeze Rotation	Stops the Rigidbody 2D rotating around the Z axes selectively.

Body Type: Kinematic



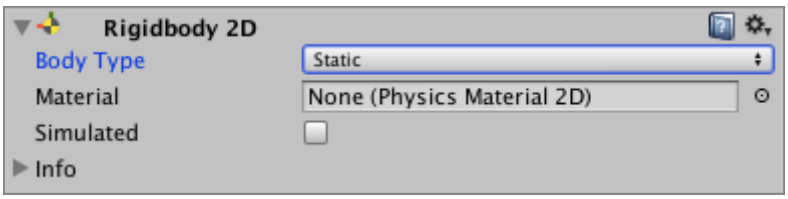
A **Kinematic** Rigidbody 2D is designed to move under simulation, but only under very explicit user control. While a **Dynamic** Rigidbody 2D is affected by gravity and forces, a **Kinematic** Rigidbody 2D isn't. For this reason, it is fast and has a lower demand on system resources than a **Dynamic** Rigidbody 2D. **Kinematic** Rigidbody 2D is designed to be repositioned explicitly via Rigidbody2D.MovePosition or Rigidbody2D.MoveRotation. Use physics queries to detect collisions, and scripts to decide where and how the Rigidbody 2D should move.

A **Kinematic** Rigidbody 2D does still move via its velocity, but the velocity is not affected by forces or gravity. A **Kinematic** Rigidbody 2D does not collide with other **Kinematic** Rigidbody 2Ds or with **Static** Rigidbody 2Ds; it only collides with **Dynamic** Rigidbody 2Ds. Similar to a **Static** Rigidbody 2D (see below), a **Kinematic** Rigidbody 2D behaves like an immovable object (as if it has infinite mass) during collisions. Mass-related properties are not available with this Body Type.

Property:	**Function:**
Body Type	Set the RigidBody 2D's component settings, so that you can manipulate movement (position and rotation) behavior and Collider 2D interaction. Options are: **Dynamic**, **Kinematic**, **Static**
Material	Use this to specify a common material for all Collider 2Ds attached to a specific parent Rigidbody 2D. **Note:** A Collider 2D uses its own Material property if it has one set. If there is no Material specified here or in the Collider 2D, the default option is **None (Physics Material 2D)**. This uses a default Material which you can set in the Physics 2D Settings window. A Collider 2D uses the following order of priority to determine which **Material** setting to use: 1. A Physics Material 2D specified on the Collider 2D itself. 2. A Physics Material 2D specified on the attached Rigidbody 2D. A Physics Material 2D default material specified in the Physics 2D Settings. **TIP:** Use this to ensure that all Collider 2Ds attached to the same **Static** Body Type Rigidbody 2D can all use the same Material.
Simulated	Enable **Simulated** (check the box) if you want the Rigidbody 2D and any attached Collider 2Ds and Joint 2Ds to interact with the physics simulation during run time. If this is disabled (the box is unchecked), these components do not interact with the simulation. See Rigidbody 2D properties: Simulated, below, for more details. This box is checked by default.

Property:	Function:
Use Full Kinematic Contacts	Enable this setting (check the box) if you want the **Kinematic** Rigidbody 2D to collide with all Rigidbody 2D Body Types. This is similar to a **Dynamic** Rigidbody 2D, except the **Kinematic** Rigidbody 2D is not moved by the physics engine when contacting another Rigidbody 2D component; instead it acts as an immovable object, with infinite mass. When **Use Full Kinematic Contacts** is disabled, the **Kinematic** Rigidbody 2D only collides with **Dynamic** Rigidbody 2Ds. See Rigidbody 2D properties: Use Full Kinematic Contacts, below, for more details. This box is unchecked by default.
Collision Detection	Define how collisions between Collider 2D are detected.
Discrete	When you set the **Collision Detection** to **Discrete**, GameObjects with Rigidbody 2Ds and Collider 2Ds can overlap or pass through each other during a physics update, if they are moving fast enough. Collision contacts are only generated at the new position.
Continuous	When the **Collision Detection** is set to **Continuous**, GameObjects with Rigidbody 2Ds and Collider 2Ds do not pass through each other during an update. Instead, Unity calculates the first impact point of any of the Collider 2Ds, and moves the GameObject there. Note that this takes more CPU time than **Discrete**.
Sleeping Mode	Define how the GameObject "sleeps" to save processor time when it is at rest.
Never Sleep	Sleeping is disabled (this should be avoided where possible, as it can impact system resources).
Start Awake	GameObject is initially awake.
Start Asleep	GameObject is initially asleep but can be woken by collisions.
Interpolate	Define how the GameObject's movement is interpolated between physics updates (useful when motion tends to be jerky).
None	No movement smoothing is applied.
Interpolate	Movement is smoothed based on the GameObject's positions in previous frames.
Extrapolate	Movement is smoothed based on an estimate of its position in the next frame.
Constraints	Define any restrictions on the Rigidbody 2D's motion.
Freeze Position	Stops the Rigidbody 2D moving in the world's x & y axes selectively.
Freeze Rotation	Stops the Rigidbody 2D rotating around the world's z axis selectively.

Body Type: Static



A **Static** Rigidbody 2D is designed to not move under simulation at all; if anything collides with it, a **Static** Rigidbody 2D behaves like an immovable object (as though it has infinite mass). It is also the least resource-

intensive body type to use. A **Static** body only collides with **Dynamic** Rigidbody 2Ds. Having two **Static** Rigidbody 2Ds collide is not supported, since they are not designed to move.

Only a very limited set of properties are available for this Body Type.

Property:	Function:
Body Type	Set the RigidBody 2D's component settings, so that you can manipulate movement (position and rotation) behavior and Collider 2D interaction. Options are: **Dynamic**, **Kinematic**, **Static**
Material	Use this to specify a common material for all Collider 2Ds attached to a specific parent Rigidbody 2D. **Note:** A Collider 2D uses its own Material property if it has one set. If there is no Material specified here or in the Collider 2D, the default option is **None (Physics Material 2D)**. This uses a default Material which you can set in the Physics 2D Settings window. A Collider 2D uses the following order of priority to determine which **Material** setting to use: 1. A Physics Material 2D specified on the Collider 2D itself. 2. A Physics Material 2D specified on the attached Rigidbody 2D. A Physics Material 2D default material specified in the Physics 2D Settings. **TIP:** Use this to ensure that all Collider 2Ds attached to the same **Static** Body Type Rigidbody 2D can all use the same Material.
Simulated	Enable **Simulated** (check the box) if you want the Rigidbody 2D and any attached Collider 2Ds and Joint 2Ds to interact with the physics simulation during run time. If this is disabled (the box is unchecked), these components do not interact with the simulation. See Rigidbody 2D properties: Simulated, below, for more details. This box is checked by default.

There are two ways to mark a Rigidbody 2D as **Static**:

For the GameObject with the Collider 2D component not to have a Rigidbody 2D component at all. All such Collider 2Ds are internally considered to be attached to a single hidden **Static** Rigidbody 2D component.

For the GameObject to have a Rigidbody 2D and for that Rigidbody 2D to be set to **Static**.

Method 1 is a shorthand for making **Static** Collider 2Ds. When creating large numbers of **Static** Collider 2Ds, it is easier not to have to add a Rigidbody 2D for each GameObject with a Collider 2D.

Method 2 exists for performance reasons. If a **Static** Collider 2D needs to be moved or reconfigured at run time, it is faster to do so when it has its own Rigidbody 2D. If a group of Collider 2Ds needs to be moved or reconfigured at run time, it is faster to have them all be children of one parent Rigidbody 2D marked as **Static** than to move each GameObject individually.

Note: As stated above, **Static** Rigidbody 2Ds are designed not to move, and collisions between two **Static** Rigidbody **2D objects** that intersect are not registered. However, **Static** Rigidbody 2Ds and **Kinematic** Rigidbody 2Ds will interact with each other if one of their Collider 2Ds is set to be a trigger. There is also a feature that changes what a **Kinematic** body will interact with (see Use Full Kinematic Contacts, below).

Rigidbody 2D properties

Simulated

Use the **Simulated** property to stop (unchecked) and start (checked) a Rigidbody 2D and any attached Collider 2Ds and Joint 2Ds from interacting with the 2D physics simulation. Changing this property is much more memory and processor-efficient than enabling or disabling individual Collider 2D and Joint 2D components.

When the **Simulated** box is checked, the following occurs:

The Rigidbody 2D moves via the simulation (gravity and physics forces are applied)
Any attached Collider 2Ds continue creating new contacts and continuously re-evaluate contacts
Any attached Joint 2Ds are simulated and constrain the attached Rigidbody 2D
All internal physics objects for Rigidbody 2D, Collider 2D & Joint 2D stay in memory

When the **Simulated** box is unchecked, the following occurs:

The Rigidbody 2D is not moved by the simulation (gravity and physics forces are not applied)
The Rigidbody 2D does not create new contacts, and any attached Collider 2D contacts are destroyed
Any attached Joint 2Ds are not simulated, and do not constrain any attached Rigidbody 2Ds
All internal physics objects for Rigidbody 2D, Collider 2D and Joint 2D are left in memory

Why is unchecking Simulated more efficient than individual component controls?

In the 2D physics simulation, a Rigidbody 2D component controls the position and rotation of attached Collider 2D components, and allows Joint 2D components to use these positions and rotations as anchor points. A Collider 2D moves when the Rigidbody 2D it is attached to moves. The Collider 2D then calculates contacts with other Collider 2Ds attached to other Rigidbody 2Ds. Joint 2Ds also constrain Rigidbody 2D positions and rotations. All of this takes simulation time.

You can stop and start individual elements of the 2D physics simulation by enabling and disabling components individually. You can do this on both Collider 2D and Joint 2D components. However, enabling and disabling individual elements of the physics simulations has memory use and processor power costs. When elements of the simulation are disabled, the 2D physics engine doesn't produce any internal physics-based objects to simulate. When elements of the simulation are enabled, the 2D physics engine does have internal physics-based objects to simulate. Enabling and disabling of 2D physics simulation components means internal GameObjects and physics-based components have to be created and destroyed; disabling the simulation is easier and more efficient than disabling individual components.

NOTE: When a Rigidbody 2D's **Simulated** option is unchecked, any attached Collider 2D is effectively 'invisible', that is; it cannot be detected by any physics queries, such as Physics.Raycast.

Use Full Kinematic Contacts

Enable this setting (check the checkbox) if you want the **Kinematic** Rigidbody 2D to collide with all Rigidbody 2D Body Types. This is similar to a **Dynamic** Rigidbody 2D, except the **Kinematic** Rigidbody 2D is not moved by the physics engine when contacting another Rigidbody 2D; it acts as an immovable object, with infinite mass.

When this setting is disabled (unchecked), a **Kinematic** Rigidbody 2D only collides with **Dynamic** Rigidbody 2Ds; it does not collide with other **Kinematic** Rigidbody 2Ds or **Static** Rigidbody 2Ds (note that trigger colliders are an exception to this rule). This means that no collision scripting callbacks (OnCollisionEnter, OnCollisionStay, OnCollisionExit) occur.

This can be inconvenient when you are using physics queries (such as Physics.Raycast) to detect where and how a Rigidbody 2D should move, and when you require multiple **Kinematic** Rigidbody 2Ds to interact with each other. Enable **Use Full Kinematic Contacts** to make **Kinematic** Rigidbody 2D components interact in this way.

Use Full Kinematic Contacts allows explicit position and rotation control of a **Kinematic** Rigidbody 2D, but still allows full collision callbacks. In a set-up where you need explicit control of all Rigidbody 2Ds, use **Kinematic** Rigidbody 2Ds in place of **Dynamic** Rigidbody 2Ds to still have full collision callback support.

Collider 2D

Leave feedback

Collider 2D components define the shape of a 2D **GameObject** for the purposes of physical **collisions**. A **Collider**, which is invisible, need not be the exact same shape as the **GameObject**'s **Mesh**; in fact, a rough approximation is often more efficient and indistinguishable in gameplay.

Colliders for 2D GameObjects all have names ending "2D". A Collider that does not have "2D" in the name is intended for use on a 3D GameObject. Note that you can't mix 3D GameObjects and 2D Colliders, or 2D GameObjects and 3D Colliders.

The Collider 2D types that can be used with **Rigidbody** 2D are:

Circle Collider 2D for circular collision areas.
Box Collider 2D for square and rectangle collision areas.
Polygon Collider 2D for freeform collision areas.
Edge Collider 2D for freeform collision areas and areas which aren't completely enclosed (such as rounded convex corners).
Capsule Collider 2D for circular or lozenge-shaped collision areas.
Composite Collider 2D for merging **Box Collider** 2Ds and Polygon Collider 2Ds.

Use Auto Mass

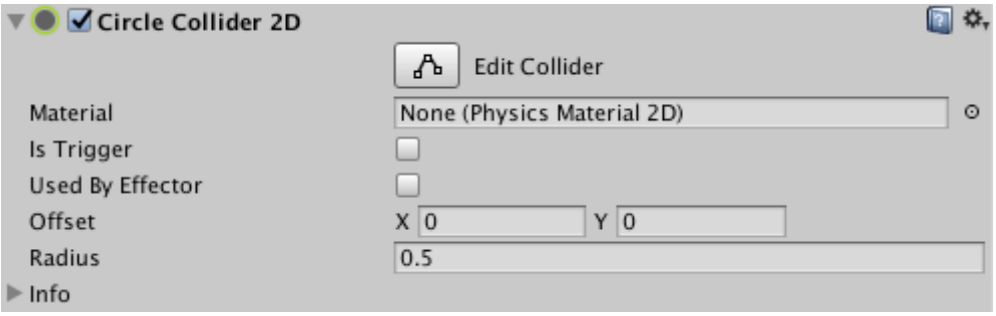
On the Rigidbody 2D component, tick the **Use Auto Mass** checkbox to automatically set the Rigidbody 2D's mass to the same value as the Collider 2D's mass. This is particularly useful in conjunction with the Buoyancy Effector 2D.

Circle Collider 2D

Leave feedback

SWITCH TO SCRIPTING

The **Circle** Collider__ 2D__ class is a **collider** for use with 2D physics. The **collider**'s shape is a circle with a defined position and radius in the local coordinate space of a **Sprite**.



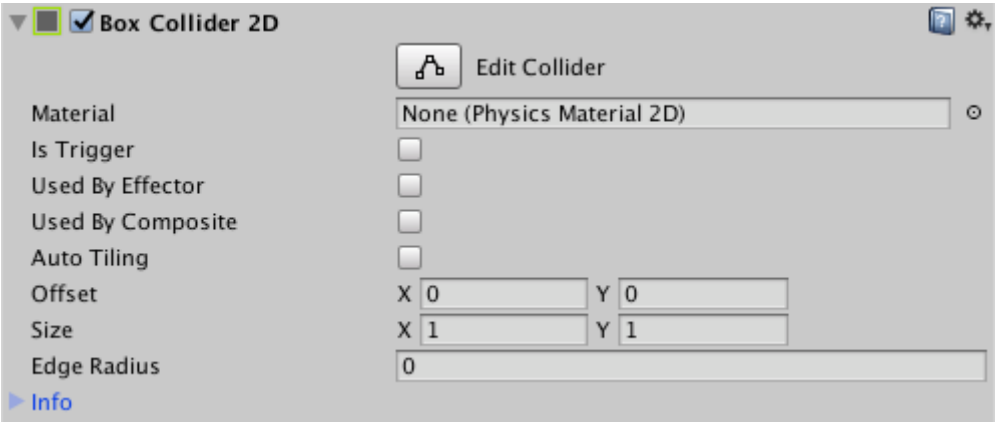
Property:	*Function:*
Material	A physics material that determines properties of **collisions**, such as friction and bounce.
Is Trigger	Check this if you want the Circle Collider 2D to behave as a trigger.
Used by Effector	Check this if you want the Circle Collider 2D to be used by an attached Effector 2D.
Offset	The local offset of the Circle Collider 2D geometry.
Radius	Radius of the circle in local space units.

Box Collider 2D

Leave feedback

SWITCH TO SCRIPTING

The **Box** Collider__An invisible shape that is used to handle physical collisions for an object. A collider doesn't need to be exactly the same shape as the object's mesh - a rough approximation is often more efficient and indistinguishable in gameplay. More info
See in Glossary 2D__ **component** is a **Collider** for use with 2D physics. Its shape is a rectangle with a defined position, width and height in the local coordinate space of a **Sprite**. Note that the rectangle is axis-aligned - that is, its edges are parallel to the X or Y axes of local space.



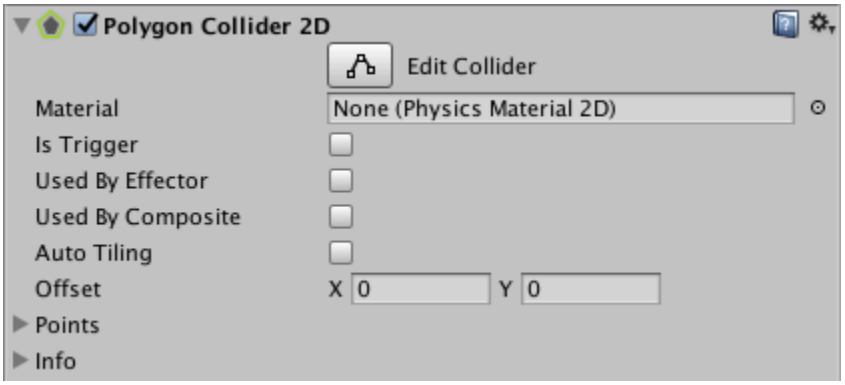
Property	*Function*
Material	A physics Material that determines properties of **collisions**, such as friction and bounce.
Is Trigger	Check this box if you want the **Box Collider** 2D to behave as a trigger.
Used by Effector	Check this box if you want the Box Collider 2D to be used by an attached Effector 2D component.
Used by Composite	Tick this checkbox if you want this Collider to be used by an attached Composite Collider 2D. When you enable **Used by Composite**, other properties disappear from the Box Collider 2D component, because they are now controlled by the attached Composite Collider 2D. The properties that disappear from the Box Collider 2D are **Material**, **Is Trigger**, **Used By Effector**, and **Edge Radius**.
Auto Tiling	Tick this checkbox if the Sprite Renderer component for the selected Sprite has the **Draw Mode** set to **Tiled**. This enables automatic updates to the shape of the Collider 2D, meaning that the shape is automatically readjusted when the Sprite's dimensions change. If you don't enable **Auto Tiling**, the Collider 2D geometry doesn't automatically repeat.
Offset	Set the local offset of the Collider 2D geometry.
Size	Set the size of the box in local space units.
Edge Radius	Controls a radius around edges, so that vertices are circular. This results in a larger Collider 2D with rounded convex corners. The default value for this setting is **0** (no radius).

Polygon Collider 2D

Leave feedback

SWITCH TO SCRIPTING

The **Polygon** Collider__An invisible shape that is used to handle physical collisions for an object. A collider doesn't need to be exactly the same shape as the object's mesh - a rough approximation is often more efficient and indistinguishable in gameplay. More info
See in Glossary 2D__ **component** is a **Collider** for use with 2D physics. The **Collider**'s shape is defined by a freeform edge made of line segments, so you can adjust it to fit the shape of the **Sprite** graphic with great precision. Note that this **Collider**'s edge must completely enclose an area (unlike the similar Edge Collider 2D).



Property	*Function*
Material	A physics material that determines properties of **collisions**, such as friction and bounce.
Is Trigger	Tick this box if you want the Collider to behave as a trigger.
Used by Effector	Whether the Collider is used by an attached effector or not.
Used by Composite	Tick this checkbox if you want this Collider to be used by an attached Composite Collider 2D. When you enable **Used by Composite**, other properties disappear from the Polygon Collider 2D component, because they are now controlled by the attached Composite Collider 2D. The properties that disappear from the Box Collider 2D are **Material**, **Is Trigger**, **Used By Effector**, and **Edge Radius**.
Used by Collider	Check this box if you want the **Box Collider** 2D to be used by an attached Composite Collider 2D component.
Auto Tiling	Tick this checkbox if you have set the **Draw Mode** of the **Sprite Renderer** component to **Tiled**. This enables automatic updates to the shape of the Collider 2D, meaning that the shape is automatically readjusted when the Sprite's dimensions change. If you don't enable **Auto Tiling**, the Collider 2D stays the same shape and size, even when the Sprite's dimensions change.
Offset	The local offset of the Collider geometry.
Points	Non-editable information about the complexity of the generated Collider.

Details

The Collider can be edited manually but it is often more convenient to let Unity determine the shape automatically. You can do this by dragging a Sprite **Asset** from the **Project view** onto the Polygon Collider 2D component in the **Inspector**.

You can edit the polygon's shape by pressing the **Edit Collider** button in the Inspector. You can exit Collider edit mode by pressing the **Edit Collider** button again. While in edit mode, you can move an existing vertex by dragging when the mouse is over that vertex. If you shift-drag while the mouse is over an edge, a new vertex will be created at the mouse location. You can remove a vertex by holding down the ctrl/cmd key while clicking on it.

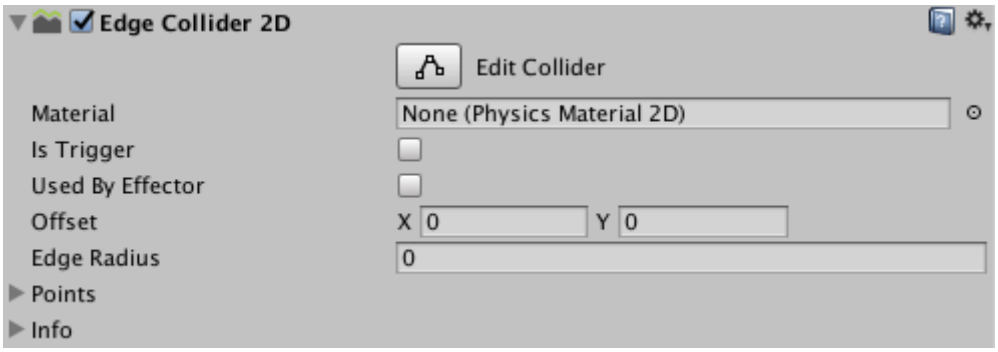
Note that you can hide the outline of the 2D move **Gizmo** while editing the Collider - just click the fold-out arrow on the Sprite Renderer component in the Inspector to collapse it.

Edge Collider 2D

Leave feedback

SWITCH TO SCRIPTING

The **Edge** Collider__An invisible shape that is used to handle physical collisions for an object. A collider doesn't need to be exactly the same shape as the object's mesh - a rough approximation is often more efficient and indistinguishable in gameplay. More info
See in Glossary 2D__ **component** is a **Collider** for use with 2D physics. The **Collider**'s shape is defined by a freeform edge made of line segments, so you can adjust it to fit the shape of the **Sprite** graphic with great precision. Note that this **Collider**'s edge need not completely enclose an area (unlike the similar Polygon Collider 2D), and for example can be a straight line or an L-shape.



Property	*Function*
Material	A physics material that determines properties of **collisions**, such as friction and bounce.
Is Trigger	Tick this box if you want the Collider to behave as a trigger.
Used by Effector	Whether the Collider is used by an attached effector or not.
Offset	The local offset of the Collider geometry.
Edge Radius	Controls a radius around edges, so that vertices are circular. This results in a larger Collider 2D with rounded convex corners. The default value for this setting is **0** (no radius).
Points	Non-editable information about the complexity of the generated collider.

Details

To edit the polyline directly, hold down the Shift key while you move the mouse over an edge or vertex in the **Scene view**. To move an existing vertex, hold down the Shift key and drag that vertex. To create a new vertex, hold down the Shift key and click where you want the vertex to be created. To remove a vertex, hold down the Ctrl (Windows) or Cmd (macOS) key and click on it.

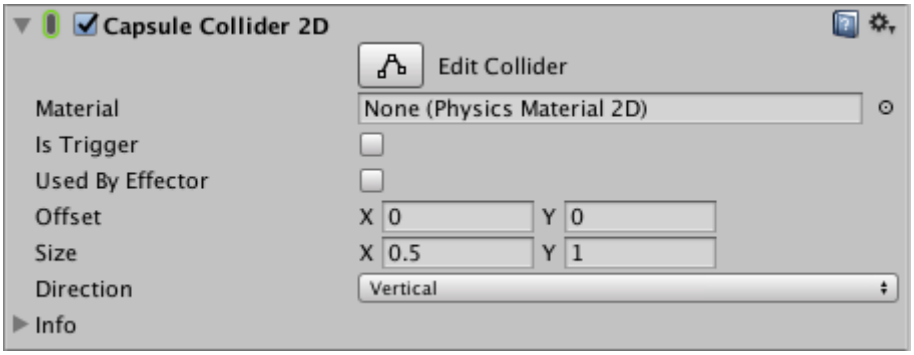
To hide the outline of the 2D move **Gizmo** while editing the Collider, click the fold-out arrow on the **Sprite Renderer** component in the **Inspector** to collapse it.

Capsule Collider 2D

Leave feedback

SWITCH TO SCRIPTING

The Capsule **Collider** 2D **component** is a 2D physics primitive that you can elongate in either a vertical or horizontal direction. The capsule shape has no vertex corners; it has a continuous circumference, which doesn't get easily caught on other **collider** corners. The capsule shape is solid, so any other **Collider** 2Ds that are fully inside the capsule are considered to be in contact with the capsule and are forced out of it over time.

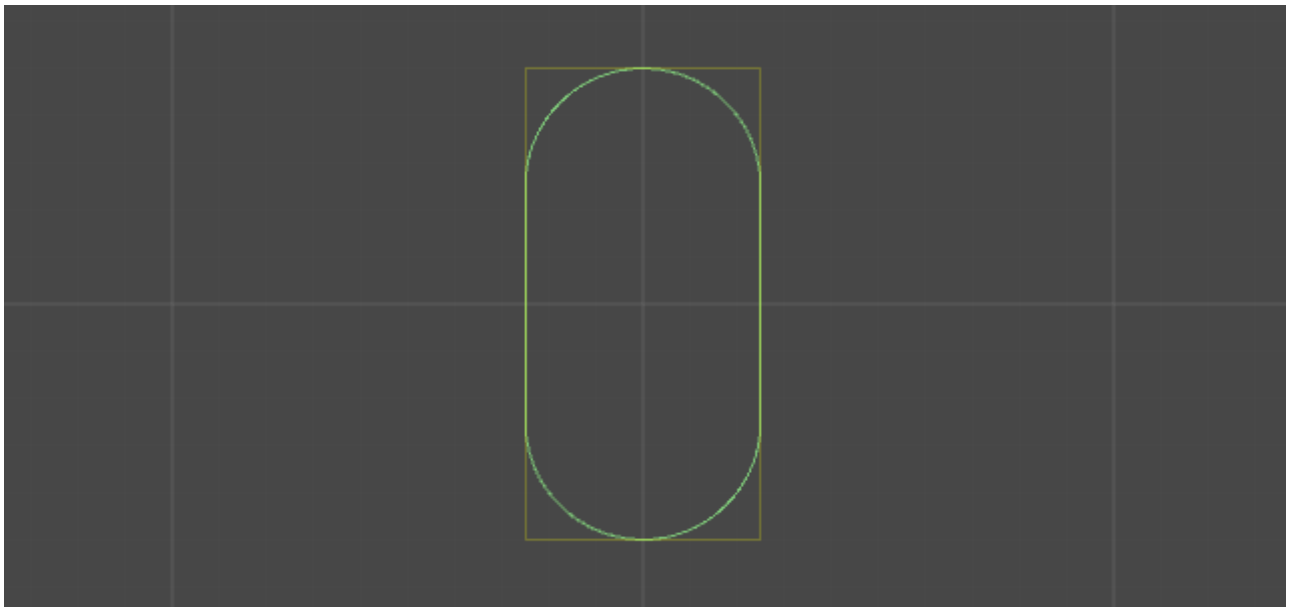


Property:	Function:
Material	Use this to define the physics material used by the **Capsule Collider** 2D. This overrides any **Rigidbody** 2D or global physics collider.
Is Trigger	Check this box to specify that the Capsule Collider 2D triggers events. If you check this box, the **physics engine** ignores this collider.
Used by Effector	Check this box to specify that an attached Effector uses this Capsule Collider 2D.
Offset	Use this to set the local offset of the Capsule Collider 2D geometry.
Size	Use this to define a box size. This box defines the region that the Capsule Collider 2D fills.
Direction	Set this to either Vertical or Horizontal. This controls which way round the capsule sets: specifically, it defines the positioning of the semi-circular end-caps.

The settings that define the Capsule Collider 2D are **Size** and **Direction**. Both the Size and Direction properties refer to **X** and **Y** (horizontal and vertical, respectively) in the local space of the Capsule Collider 2D, and not in world-space.

A typical way to set up the Capsule Collider 2D is to set the **Size** to match the **Direction**. For example, if the Capsule Collider 2D's **Direction** is **Vertical**, the **Size** of **X** is 0.5 and the **Size** of **Y** is 1, this makes the vertical direction capsule taller, rather than wider.

In the example below, the **X** and **Y** are represented by the yellow lines.

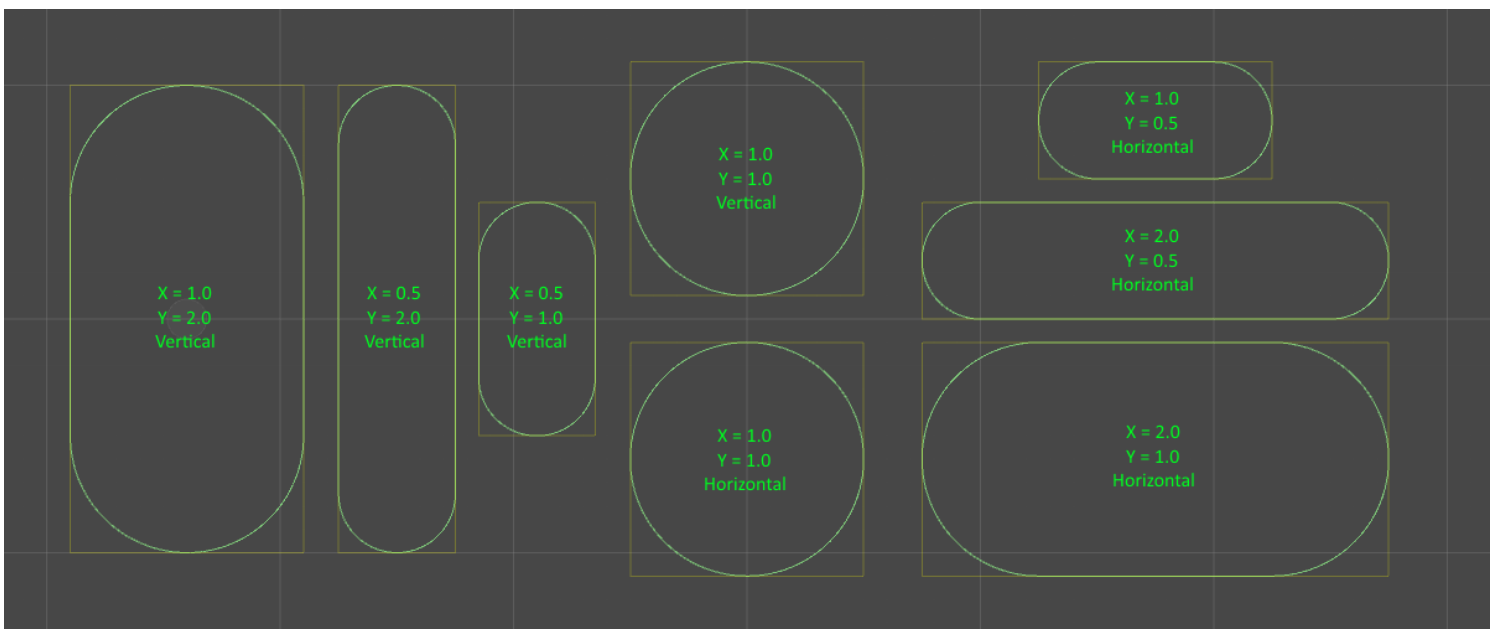


An example of a Capsule Collider 2D set so the **Size** matches **Direction**

Capsule configuration examples

You can change the Capsule Collider 2D with different configurations. Below are some examples.

Note that when the **X** and **Y** of the **Size** property are the same, the Capsule Collider 2D always approximates to a circle.



Examples of Capsule Collider 2D configurations

Tip

A known issue in physics engines (including Box 2D) is that moving across multiple colliders, even colliders that are perfectly aligned numerically, causes one or both of the colliders to register a **collision** between the two colliders. This can cause the collider to slow down or stop.

While the Capsule Collider 2D can help reduce this problem, it isn't a solution to it. A better solution is to use a single collider for a surface; for example, the Edge Collider 2D.

Composite Collider 2D

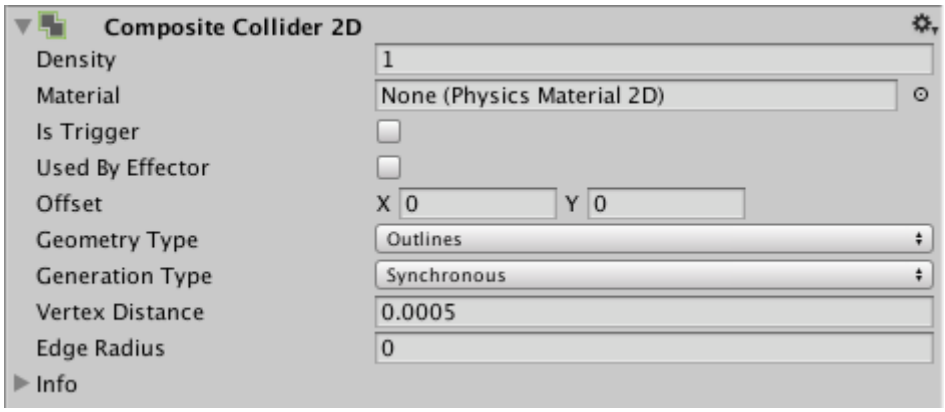
Leave feedback

SWITCH TO SCRIPTING

The Composite Collider 2D component is a Collider for use with 2D physics. Unlike most colliders, it does not define an inherent shape. Instead, it merges the shapes of any Box Collider 2D or Polygon Collider 2D that you set it up to use. The Composite Collider 2D uses the vertices (geometry) from any of these Colliders, and merges them together into new geometry controlled by the Composite Collider 2D itself.

The **Box Collider** 2D and Polygon Collider 2D components have a **Used By Composite** checkbox. Tick this checkbox to attach them to the Composite Collider 2D. These Colliders also have to be attached to the same Rigidbody 2D as the Composite Collider 2D. When you enable **Used by Composite**, other properties disappear from that component, because they are now controlled by the attached Composite Collider 2D.

See API documentation on Composite Collider 2D for more information about scripting with the Composite Collider 2D.



Property	Function
Density	Change the density to change the mass calculations of the GameObject's associated **Rigidbody** 2D. If you set the value to 0, its associated **Rigidbody** 2D ignores the Collider 2D for all mass calculations, including centre of mass calculations. Note that this option is only available if you have enabled **Use Auto Mass** in the associated **Rigidbody** 2D component.
Material	A Physics Material 2D that determines the properties of **collisions**, such as friction and bounce.
Is Trigger	Check this box if you want the Composite Collider 2D to behave as a trigger (see overview documentation on Colliders to learn more about triggers).
Used by Effector	Check this box if you want the Composite Collider 2D to be used by an attached Effector 2D component.
Offset	Set the local offset of the Collider 2D geometry.
Geometry Type	When merging Colliders, the vertices from the selected Colliders are composed into one of two different geometry types. Use this drop-down to set the geometry type to either **Outlines** or **Polygons**.
Outlines	Produces a Collider 2D with hollow outlines, identical to what the Edge Collider 2D produces.

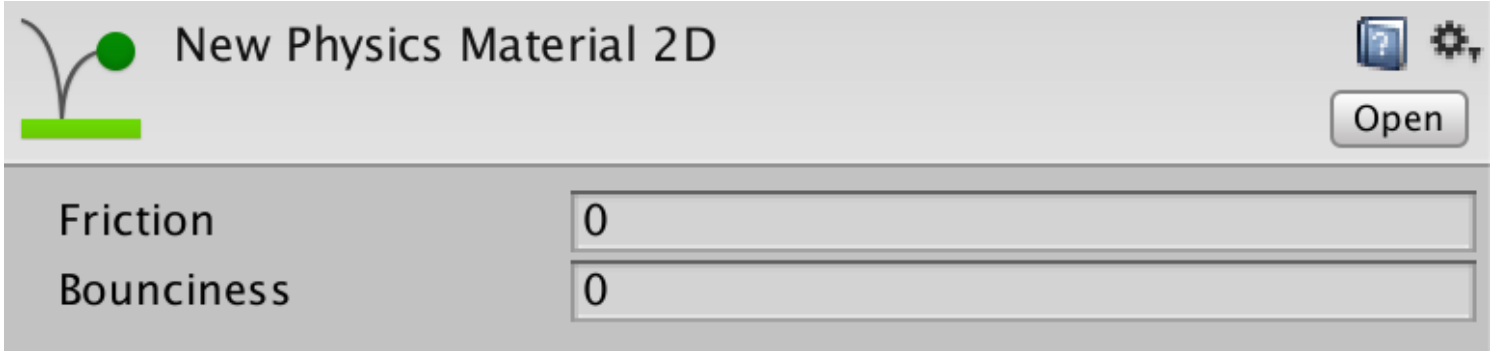
Property	Function
Polygons	Produces a Collider 2D with solid polygons, identical to what the Polygon Collider 2D produces.
Generation Type	The method used to control when geometry is generated when either the Composite Collider 2D is changed, or any of the Colliders it is composing is changed.
Synchronous	When a change is made to the Composite Collider 2D or any of the colliders it is using, Unity generates new geometry immediately.
Manual	New geometry generation happens only when you request it. To request generation, either call the CompositeCollider2D.GenerateGeometry script API, or click the **Regenerate Geometry** button that appears under the selection.
Vertex Distance	Set a value for the minimum spacing allowed for any vertices gathered from Colliders being composed. Any vertex closer than this limit is removed. This allows control over the effective resolution of the vertex compositing.
Edge Radius	Controls a radius around edges, so that vertices are circular. This results in a larger Collider 2D with rounded convex corners. The default value for this setting is 0 (no radius). This only works when the **Geometry Type** is set to **Outlines**.

Physics Material 2D

Leave feedback

SWITCH TO SCRIPTING

A **Physics Material 2D** is used to adjust the friction and bounce that occurs between 2D physics objects when they collide. You can create a Physics Material 2D from the Assets menu (**Assets > Create > Physics Material 2D**).



Properties

Property:	*Function:*
Friction	Coefficient of friction for this **collider**.
Bounciness	The degree to which **collisions** rebound from the surface. A value of 0 indicates no bounce while a value of 1 indicates a perfect bounce with no loss of energy.

Details

To use a Physics Material 2D, simply drag it onto an object with a 2D collider attached or drag it to the collider **component** in the **inspector**. Note that for 3D physics, the equivalent **asset** is referred to as a *Physic Material* (ie, no S at the end of *physic*) - it is important in scripting not to get the two spellings confused.

2D Joints

Leave feedback

As the name implies, **joints** attach **GameObjects** together. You can only attach 2D **joints** to **GameObjects** which have a **Rigidbody** 2D **component** attached, or to a fixed position in world space.

2D joints all have names ending '2D'. A joint named without a '2D' ending is a 3D joint. 2D joints work with game objects in 2D and 3D joints work with game objects in 3D.

(See **Details and Hints**, below, for useful background information on all 2D joints.)

There are different types of 2D joints. See each joint reference page for detailed information about their properties:

Distance Joint 2D - attaches two game objects controlled by rigidbody physics together and keeps them a certain distance apart.

Fixed Joint 2D - keeps two objects in a position relative to each other, so the objects are always offset at a given position and angle. For example, objects that need to react as if they are rigidly connected: They can't move away from each other, they can't move closer together, and they can't rotate with respect to each other. You can also use this joint to create a less rigid connection that flexes.

Friction Joint 2D - reduces both the linear and angular velocities between two game objects controlled by rigidbody physics to zero (ie: it slows them down and stops them). For example; a platform that rotates but resists that movement.

Hinge Joint 2D- allows a game object controlled by rigidbody physics to be attached to a point in space around which it can rotate. For example; the pivot on a pair of scissors.

Relative Joint 2D - allows two game objects controlled by rigidbody physics to maintain a position based on each other's location. Use this joint to keep two objects offset from each other. For example; a space-shooter game where the player has extra gun batteries that follow them.

Slider Joint 2D - allows a game object controlled by rigidbody physics to slide along a line in space, like sliding doors, for example.

Spring Joint 2D - allows two game objects controlled by rigidbody physics to be attached together as if by a spring.

Target Joint 2D - connects to a specified target, rather than another rigid body object, as other joints do. It is a spring type joint, which you could use for picking up and moving an object acting under gravity, for example.

Wheel Joint 2D - simulates wheels and suspension.

Details and Hints

Constraints

All joints provide one or more constraints that apply to Rigidbody 2D behaviour. A constraint is a 'rule' which the joint will try to ensure isn't permanently broken. There are different types of constraints available but usually a joint will only provide a few of them, sometimes only one. Some constraints limit behaviour such as ensuring a

rigid body stays on a line, or in a certain position. Some are 'driving' constraints such as a motor that rotates or moves a rigid body object, trying to maintain a certain speed.

Temporarily breaking constraints

The physics system expects that constraints do become temporarily broken; the objects may move further apart than their distance constraint tells them, or objects may move faster than their motor-speed constraint. When a constraint isn't broken, the joint doesn't apply any forces and does little work. It is when a constraint is broken that the joint applies forces to fix the constraint: So for the 'driving' constraints mentioned above, it maintains a distance or ensures a motor-speed. This force, however, doesn't always instantaneously fix the constraint. Although it usually happens very fast, it can happen over time.

This time lag can lead to joints 'stretching' or seeming 'soft'. The lag happens because the physics system is trying to apply joint-forces to fix constraints, whilst at the same time other game physics forces are acting to break constraints. In addition to the conflicting forces acting on game objects, some joints are more stable and react faster than others.

Whatever constraints the joint provides, the joint only uses forces to fix the constraint. These are either a linear (straight line) force or angular (torque) force.

HINT: Given the conflicing forces acting on joints, it is always good to be cautious when applying large forces to rigid body objects that have joints attached, especially those with large masses.

Permanently breaking joints

All joints have the ability to stop working completely (that is break) when a force exceeds a specified limit. The limit that causes breaking due to excessive linear force is called the "break force". The limit that causes breaking due to excessive torque force is called the "break torque".

If a joint applies linear force, then it has a **Break Force** option.
If a joint applies an angular (rotation) force then it has a **Break Torque** option.
Both these limits are pre-set to **Infinity**: This means that they have no limit.

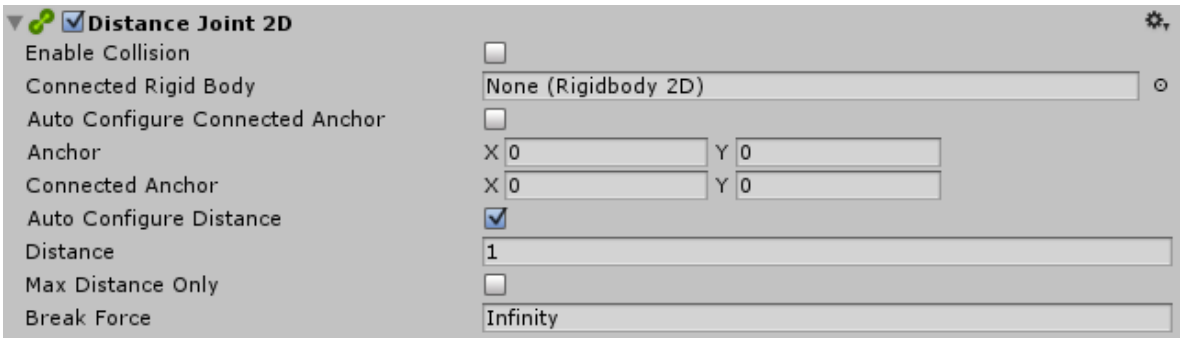
When a **Break Force** or **Break Torque** limit is exceeded, the joint breaks and the component deletes itself from its GameObject.

Distance Joint 2D

Leave feedback

SWITCH TO SCRIPTING

Distance Joint__ 2D__ is a 2D **joint** that attaches two **GameObjects** controlled by **Rigidbody 2D** physics, and keeps them a certain distance apart.



Property:	*Function:*
Enable Collision	Check this box to enable **collision** between the two connected GameObjects.
Connected Rigid Body	Use this field to specify the other GameObject that this Distance Joint 2D connects to. If ths is left as **None (Rigidbody 2D)**, the other end of the Distance Joint 2D is fixed at a point in space defined by the **Connected Anchor** setting. Select the circle to the right of the field to view a list of GameObjects to connect to.
Auto Configure Connected Anchor	Check this box to automatically set the anchor location for the other GameObject this Distance Joint 2D connects to. If you enable this, you don't need to complete the **Connected Anchor** fields.
Anchor	Define where (in terms of x, y co-ordinates on the **Rigidbody 2D**) the end point of the Distance Joint 2D connects to this GameObject.
Connected Anchor	Define where (in terms of x, y co-ordinates on the **Rigidbody 2D**) the end point of the Distance Joint 2D connects to the other GameObject.
Auto Configure Distance	Check this box to automatically detect the current distance between the two GameObjects, and set it as the distance that the Distance Joint 2D keeps between the two GameObjects. If you enable this, you don't need to complete the **Distance** field.
Distance	Specify the distance that the Distance Joint 2D keeps between the two GameObjects.
Max Distance Only	If enabled, the Distance Joint 2D only enforces a maximum distance, so the connected GameObjects can still move closer to each other, but not further than the **Distance** field defines. If this is not enabled, the distance between the GameObjects is fixed.
Break Force	Specify the force level needed to break and therefore delete the Distance Joint 2D. **Infinity** means it is unbreakable.

Notes

The aim of this Joint 2D is to keep distance between two points. Those two points can be two **Rigidbody** 2D components or a **Rigidbody** 2D **component** and a fixed position in the world. To connect a **Rigidbody 2D** component to a fixed position in the world, set the **Connected Rigidbody** field to None.

This Joint 2D does not apply torque, or rotation. It does apply a linear force to both connected items, using a very stiff, simulated spring to maintain the distance. You cannot configure the spring.

This Joint 2D has a selectable constraint:

Constraint A: Maintains a fixed distance between two anchor points on two bodies (when **Max Distance Only** is unchecked).
Constraint B: Maintains maximum distance only between two anchor points on two bodies (when **Max Distance Only** is checked).
You can use this Joint 2D to construct physical objects that need to behave as if they are connected with a rigid connection that can rotate.

Using constraint A (**Max Distance Only** unchecked), you can create a fixed length connection, such as two wheels on a bicycle.
Using constraint B (**Max Distance Only** checked), you can create a constrained length connection, such as a yo-yo moving towards and away from a hand.
See Joints 2D: *Details and Hints* for useful background information on all 2D Joints.

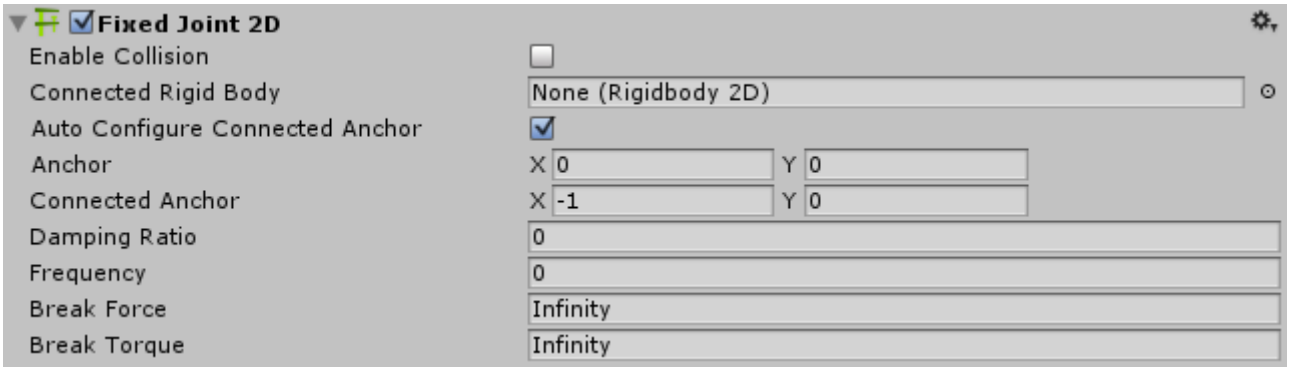
Fixed Joint 2D

Leave feedback

SWITCH TO SCRIPTING

Apply this **component** to two **GameObjects** controlled by **Rigidbody** 2D physics to keep them in a position relative to each other, so the **GameObjects** are always offset at a given position and angle. It is a spring-type 2D **joint** for which you don't need to set maximum forces. You can set the spring to be rigid or soft.

See *Fixed Joint 2D and Relative Joint 2D* below for more information about the differences between **Fixed Joint 2D** and **Relative Joint 2D**.



Property:	*Function:*
Enable Collision	Check this box to enable the ability for the two connected GameObjects to collide with each other.
Connected Rigid Body	Specify the other GameObject this Fixed Joint 2D connects to. Leave this as **None** to fix the other end of the Fixed Joint 2D to a point in space defined by the **Connected Anchor** setting. Select the circle to the right of the field to view a list of GameObjects to connect to.
Auto Configure Connected Anchor	Check this box to automatically set the anchor location for the other GameObject this Fixed Joint 2D connects to. If you check this, you don't need to complete the **Connected Anchor** fields.
Anchor	The place (in terms of x,y co-ordinates on the **Rigidbody 2D**) where the end point of the joint connects to *this* object.
Connected Anchor	The place (in terms of x, y co-ordinates on the **Rigidbody 2D**) where the end point of the Fixed Joint 2D connects to the other GameObject.
Damping Ratio	Define the degree to which you want to suppress spring oscillation, between 0 and 1, the higher the value, the less movement.
Frequency	The frequency at which the spring oscillates while the GameObjects are approaching the separation distance you want, measured in cycles per second. In the range 1 to 1,000,000, the higher the value, the stiffer the spring. Note that if the frequency is set to 0, the spring is completely stiff.
Break Force	Specify the force level needed to break and therefore delete the joint. **Infinity** means it is unbreakable.
Break Torque	Specify the torque level needed to break and therefore delete the joint. **Infinity** means it is unbreakable.

Notes

The aim of this joint is to maintain a relative linear and angular offset between two points. Those two points can be two **Rigidbody 2D** components or a **Rigidbody 2D** component and a fixed position in the world. (Connect to a fixed position in the world by setting **Connected Rigidbody** to None).

The linear and angular offsets are based upon the relative positions and orientations of the two connected points, so you change the offsets by moving the connected GameObjects in your **Scene view**.

The joint applies both linear and torque forces to connected Rigidbody 2D GameObjects. It uses a simulated spring that is pre-configured to be as stiff as the simulation can provide. You can change the spring's value to make it weaker using the **Frequency** setting.

When the spring applies its force between the GameObjects, it tends to overshoot the desired distance between them and then rebound repeatedly, resulting in a continuous oscillation. The damping ratio determines how quickly the oscillation reduces and brings the GameObjects to rest. The frequency is the rate at which it oscillates either side of the target distance; the higher the frequency, the stiffer the spring.

Fixed Joint 2D has two simultaneous constraints:

Maintain the linear offset between two anchor points on two Rigidbody 2D GameObjects.
Maintain the angular offset between two anchor points on two Rigidbody 2D GameObjects.
You can use this joint to construct physical GameObjects that need to react as if they are rigidly connected. They can't move away from each other, they can't move closer together, and they can't rotate with respect to each other, such as a bridge made of sections which hold rigidly together.

You can also use this joint to create a less rigid connection that flexes - for example, a bridge made of sections which are slightly flexible.

Fixed Joint 2D and Relative Joint 2D

It is important to know the major differences between Fixed Joint 2D and Relative Joint 2D:

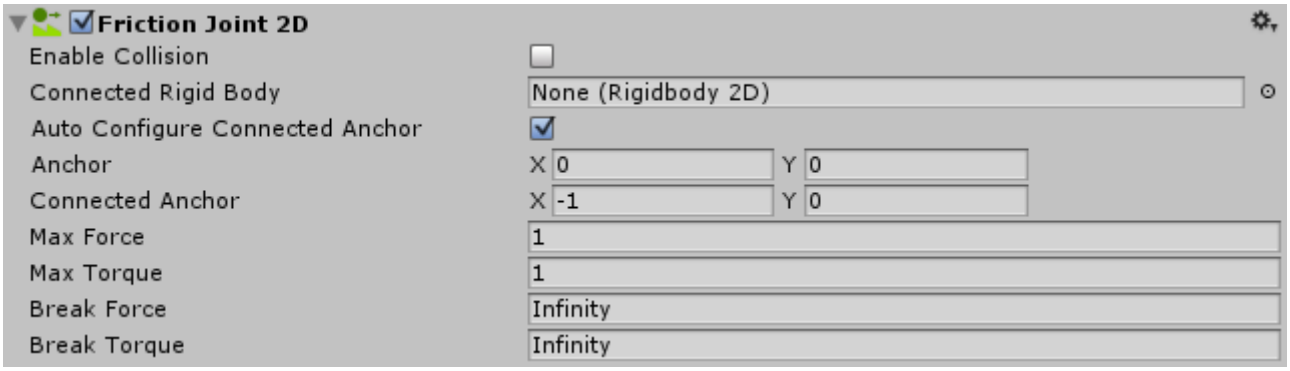
Fixed Joint 2D is a spring-type joint. **Relative Joint 2D** is a motor-type joint with a maximum force and/or torque.
Fixed Joint 2D uses a spring to maintain the relative linear and angular offsets. **Relative Joint 2D** uses a motor. You can configure a joint's spring or motor.
Fixed Joint 2D works with anchor points (it's derived from script **Anchored Joint 2D**); it maintains the relative linear and angular offset between the anchors. **Relative Joint 2D** doesn't have anchor points (it's derived directly from script **Joint 2D**).
Fixed Joint 2D cannot modify the relative linear and angular offsets in real time. **Relative Joint 2D** can.
See Joints 2D: *Details and Hints* for useful background information on all 2D joints.

Friction Joint 2D

Leave feedback

SWITCH TO SCRIPTING

The **Friction** Joint__A physics component allowing a dynamic connection between rigidbodies, usually allowing some degree of movement such as a hinge. More info
See in Glossary 2D__ connects **GameObjects** controlled by **Rigidbody** 2D physics. The Friction **Joint** 2D reduces both the linear and angular velocities between the objects to zero (ie, it slows them down). You can use this joint to simulate top-down friction, for example.



Property:	*Function:*
Enable Collision	Check the box to enable **collisions** between the two connected GameObjects.
Connected Rigid Body	Specify the other GameObject this Friction Joint 2D connects to. If you leave this as **None**, the other end of the Friction Joint 2D is fixed to a point in space defined by the **Connected Anchor** setting. Select the circle to the right of the field to view a list of GameObjects to connect to.
Auto Configure Connected Anchor	Check this box to automatically set the anchor location for the other GameObject this Friction Joint 2D connects to. If you check this, you don't need to complete the **Connected Anchor** fields.
Anchor	Define where (in terms of x, y co-ordinates on the **RigidBody 2D**) the end point of the Friction Joint 2D connects to this GameObject.
Connected Anchor	Define where (in terms of x, y co-ordinates on the **RigidBody 2D**) the end point of the Friction Joint 2D connects to the other GameObject.
Max Force	Sets the linear (or straight line) movement between joined GameObjects. A high value resists the linear movement between GameObjects.
Max Torque	Sets the angular (or rotation) movement between joined GameObjects. A high value resists the rotation movement between GameObjects.
Break Force	Specify the force level needed to break and therefore delete the Friction Joint 2D. **Infinity** means it is unbreakable.
Break Torque	Specify the torque level needed to break and therefore delete the Friction Joint 2D. **Infinity** means it is unbreakable.

Notes

Use the Friction Joint 2D to slow down movement between two points to a stop. This joint's aim is to maintain a zero relative linear and angular offset between two points. Those two points can be two **Rigidbody 2D**

components or a **Rigidbody 2D component** and a fixed position in the world. (Connect to a fixed position in the world by setting **Connected Rigidbody** to None).

Resistance

The joint applies linear force (**Force**) and angle force (**Torque**) to both Rigidbody 2D points. It uses a simulated motor that is pre-configured to have a low motor power (and so, low resistence). You can change the resistance to make it weaker or stronger.

Strong Resistance:

A high (1,000,000 is the highest) **Max Force** creates strong linear resistance. The Rigidbody 2D GameObjects won't move in a line relative to each other very much.
A high (1,000,000 is the highest) **Max Torque** creates strong angular resistance. The Rigidbody 2D GameObjects won't move at an angle relative to each each other very much.
Weak Resistance:

A low **Max Force** creates weak linear resistance. The Rigidbody 2D GameObjects move easily in a line relative to each other.
A low **Max Torque** creates weak angular resistance. The Rigidbody 2D GameObjects move easily at an angle relative to each each other.

Constraints

Friction Joint 2D has two simultaneous constraints:

Maintain a zero relative linear velocity between two anchor points on two Rigidbody 2Ds
Maintain a zero relative angular velocity between two anchor points on two Rigidbody 2Ds
You can use this joint to construct physical GameObjects that need to behave as if they have friction. They can resist either linear movement or angular movement, or both linear and angular movement. For example:

A platform that does rotate, but resists applied forces, making it difficult but possible for the player to move it.
A ball that resists linear movement. The ball's friction is related to the GameObject's velocity and not to any collisions. It acts like the **Linear Drag** and **Angular Drag** which is set in **Rigidbody 2D**. The difference is that Friction Joint 2D has the option of maximum **Force** and **Torque** settings.)
See Joints 2D: *Details and Hints* for useful background information on all 2D joints.

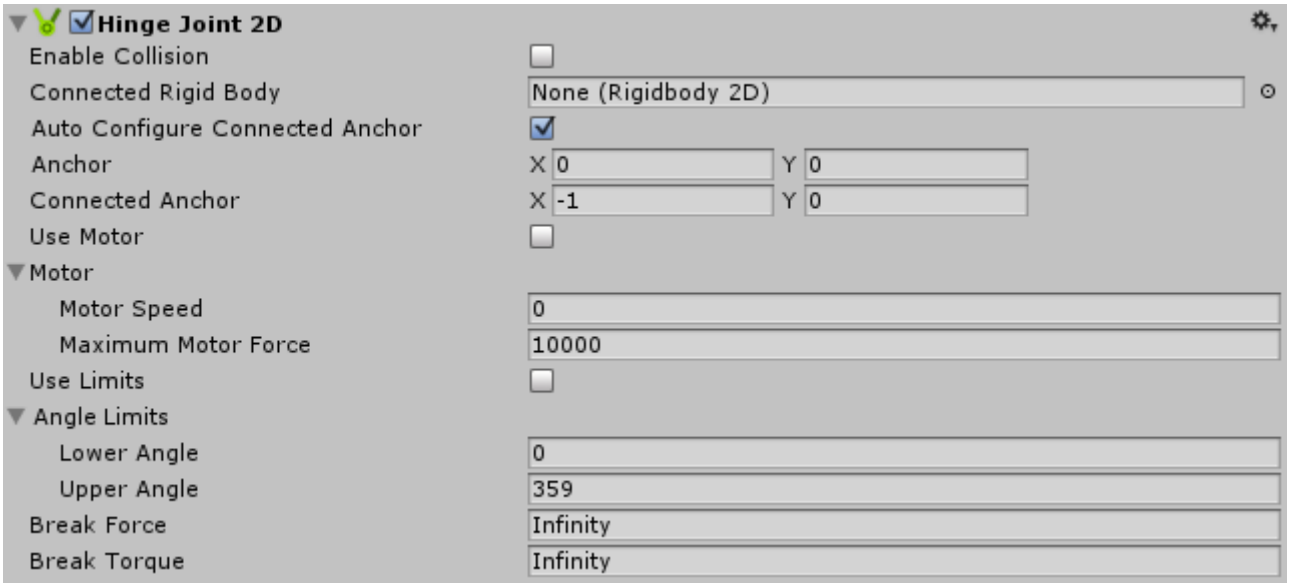
Hinge Joint 2D

Leave feedback

SWITCH TO SCRIPTING

The **Hinge** Joint__ 2D__ **component** allows a **GameObject** controlled by **RigidBody** 2D physics to be attached to a point in space around which it can rotate. The rotation can be left to happen passively (for example, in response to a collision) or can be actively powered by a motor torque provided by the Joint 2D itself. You can set limits to prevent the hinge from making a full rotation, or make more than a single rotation.

Properties



Property:	*Function:*
Enable Collision	Check the box to enable **collisions** between the two connected GameObjects.
Connected Rigid Body	Specify the other GameObject this **Hinge Joint** 2D connects to. If you leave this as **None**, the other end of the **Hinge Joint** 2D is fixed to a point in space defined by the **Connected Anchor** setting. Select the circle to the right of the field to view a list of GameObjects to connect to.
Auto Configure Connected Anchor	Check this box to automatically set the anchor location for the other GameObject this Hinge Joint 2D connects to. If you check this, you don't need to complete the **Connected Anchor** fields.
Anchor	Define where (in terms of x, y co-ordinates on the **RigidBody 2D**) the end point of the Friction Joint 2D connects to this GameObject.
Connected Anchor	Define where (in terms of x, y co-ordinates on the **RigidBody 2D**) the end point of the Friction Joint 2D connects to the other GameObject.
Motor	Use this to change Motor settings.
Use Motor	Check the box to enable the hinge motor.
Motor Speed	Set the target motor speed, in degrees per second.

Property:	*Function:*
Maximum Motor Force	Set the maximum torque (or rotation) the motor can apply while attempting to reach the target speed.
Use Limits	Check this box to limit the rotation angle
Angle Limits	Use these settings to set limits if **Use Limits** is enabled.
Lower Angle	Set the lower end of the rotation arc allowed by the limit.
Upper Angle	Set the upper end of the rotation arc allowed by the limit.
Break Force	Specify the linear (or straight line) force level needed to break and therefore delete the joint. **Infinity** means it is unbreakable.
Break Torque	Specify the the angular (or rotation) level needed to break and therefore delete the joint. **Infinity** means it is unbreakable.

Notes

The Hinge Joint 2D's name suggest a door hinge. It can be used as a door hinge, but also anything that rotates around a particular point - for example: machine parts, powered wheels, and pendulums.

You can use this joint to make two points overlap. Those two points can be two **Rigidbody 2D** components, or a **Rigidbody 2D** component and a fixed position in the world. Connect the Hinge Joint 2D to a fixed position in the world by setting **Connected Rigid Body** to None. The joint applies a linear force to both connected Rigidbody 2D GameObjects.

The Hinge Joint 2D has a simulated rotational motor which you can turn on or off. Set the **Maximum Motor Speed** and **Maximum Motor Force** to control the angular speed (**Torque**) and make the two Rigidbody 2D GameObjects rotate in an arc relative to each other. Set the limits of the arc using **Lower Angle** and **Upper Angle**.

Constraints

Hinge Joint 2D has three simultaneous constraints. All are optional:

Maintain a relative linear distance between two anchor points on two Rigidbody 2D GameObjects.
Maintain an angular speed between two anchor points on two Rigidbody 2D GameObjects (limited with a maximum torque in **Maximum Motor Force** .
Maintain an angle within a specified arc.
You can use this joint to construct physical GameObjects that need to behave as if they are connected with a rotational pivot. For example:

A see-saw pivot where the horizontal section is connected to the base. Use the joint's **Angle Limits** to simulate the highest and lowest point of the see-saw's movement.
A pair of scissors connected together with a hinge pivot. Use the joint's **Angle Limits** to simulate the closing and maximum opening of the scisssors.
A simple wheel connected to the body of a car with the pivot connecting the wheel at its center to the car. In this example you can use the Hinge Joint 2D's motor to rotate the wheel.
See Joints 2D: *Details and Hints* for useful background information on all 2D joints.

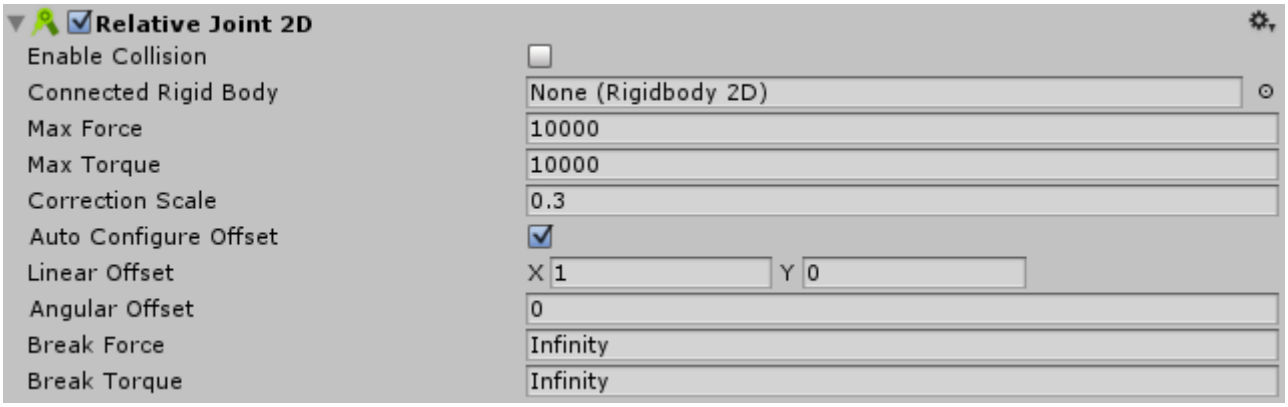
Relative Joint 2D

Leave feedback

SWITCH TO SCRIPTING

This **joint component** allows two game objects controlled by **rigidbody** physics to maintain in a position based on each other's location. Use this **joint** to keep two objects offset from each other, at a position and angle you decide.

See *Details* below for more information about the differences between **RelativeJoint2D** and **FixedJoint2D**.



Property:	*Function:*
Enable Collision	Can the two connected objects collide with each other? Check the box for yes.
Connected Rigid Body	Specify here the other object this joint connects to. Leave this as **None** and the other end of the joint will be fixed at a point in space. Select the circle to the right of the field to view a list of objects to connect to.
Max Force	Sets the linear (straight line) offset between joined objects - a high value (of up to 1,000) uses high force to maintain the offset.
Max Torque	Sets the angular (rotation) movement between joined objects - a high value (of up to 1,000) uses high force to maintain the offset.
Correction Scale	Tweaks the joint to make sure it behaves as required. (Increasing the Max Force or Max Torque may affect behaviour so that the joint doesn't reach its target, use this setting to correct it.) The default setting of 0.3 is usually appropriate but it may need tweaking between the range of 0 and 1.
Auto Configure Offset	Check this box to automatically set and maintain the distance and angle between the connected objects. (Selecting this option means you don't need to manually enter the **Linear Offset** and **Angular Offset**.)
Linear Offset	Enter local space co-ordinates to specify and maintain the distance between the connected objects.
Angular Offset	Enter local space co-ordinates to specify and maintain the angle between the connected objects.
Break Force	Specify the force level needed to break and so delete the joint. **Infinity** means it is unbreakable.
Break Torque	Specify the torque level needed to break and so delete the joint. **Infinity** means it is unbreakable.

Details

(See also Joints 2D: *Details and Hints* for useful background information on all 2D joints.)

The aim of this joint is to maintain a relative linear and angular distance (offset) between two points. Those two points can be two **Rigidbody2D** components or a **Rigidbody2D** component and a fixed position in the world. (Connect to a fixed position in the world by setting **Connected Rigidbody** to None).

The joint applies a linear and angular (torque) force to both connected rigid body objects. It uses a simulated motor that is pre-configured to be quite powerful: It has a high **Max Force** and **Max Torque** limit. You can lower these values to make the motor less powerful motor or turn-off it off completely.

This joint has two simultaneous constraints:

Maintain the specified linear offset between the two rigid body objects.
Maintain the starting angular offset between the two rigid body objects.
For Example:

You can use this joint to construct physical objects that need to:

Keep a distance apart from each other, as if they are unable to move further away from each other or closer together. (You decide the distance they are apart from each other. The distance can change in real-time.)
Rotate with respect to each other only at particular angle. (You decide the angle.)
Some uses may need the connection to be flexible, such as: A space-shooter game where the player has extra gun batteries that follow them. You can use the Relative Joint to give the trailing gun batteries a slight lag when they follow, but make them rotate with the player with no lag.

Some uses may need a configurable force, such as: A game where the the **camera** follows a player using a configurable force to keep track.

Fixed vs. Relative Joint

FixedJoint2D is spring type joint. **RelativeJoint2D** is a motor type joint with a maximum force and/or torque.

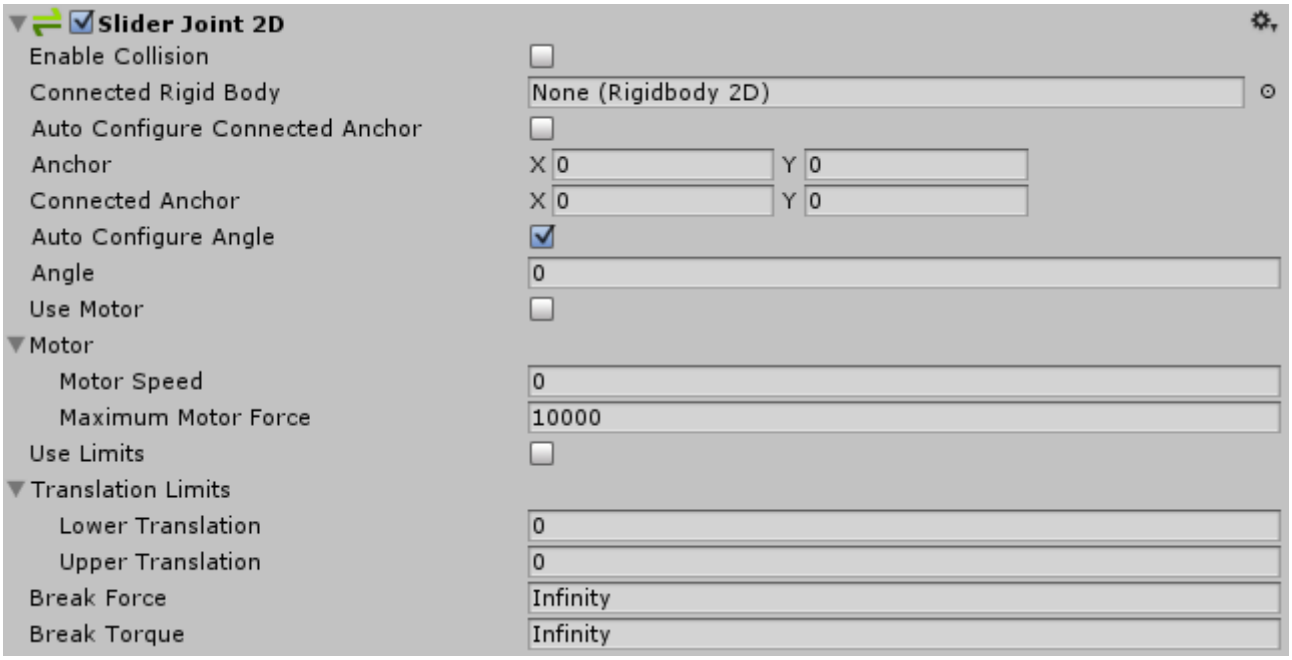
The **Fixed Joint** uses a spring to maintain the relative linear and angular offsets and the Relative joint uses a motor. You can configure a joint's spring or motor.
The Fixed joint works with anchor points (it's derived from script **AnchoredJoint2D**): It maintains the relative linear and angular offset between the anchors. The Relative joint doesn't have anchor points (it's derived directly from script **Joint2D**).
The Relative joint can modify the relative linear and angular offsets in real time: The Fixed joint cannot.

Slider Joint 2D

Leave feedback

SWITCH TO SCRIPTING

This **joint** allows a game object controlled by rigidbody physics to slide along a line in space, like sliding doors, for example. The object can freely move anywhere along the line in response to **collisions** or forces or, alternatively, it can be moved along by a motor force, with limits applied to keep its position within a certain section of the line.



Property:	*Function:*
Enable Collision	Can the two connected objects collide with each other? Check the box for yes.
Connected Rigid Body	Specify here the other object this joint connects to. Leave this as **None** and the other end of the joint will be fixed at a point in space defined by the **Connected Anchor** setting. Select the circle to the right of the field to view a list of objects to connect to.
Auto Configure Connected Anchor	Check this box to automatically set the anchor location for the other object this joint connects to. (Check this instead of completing the **Connected Anchor** fields.)
Anchor	The place (in terms of X, Y co-ordinates on the **RigidBody**) where the end point of the joint connects to *this* object.
Connected Anchor	The place (in terms of X, Y co-ordinates on the **RigidBody**) where the end point of the joint connects to *the other* object.
Auto Configure Angle	Check this box to automtically detect the angle between the two objects and set it as the angle that the joint keeps between the two objects. (By selecting this, you don't need to manually specify the angle.)
Angle	Enter the the angle that the joint keeps between the two objects.
Use Motor	Use the sliding motor? Check the box for yes.
Motor	
Motor Speed	Target motor speed (m/sec).

Property:	*Function:*
Maximum Motor Force	The maximum force the motor can apply while attempting to reach the target speed.
Use Limits	Should there be limits to the linear (straight line) force? Check the box for yes.
Translation Limits	
Lower Translation	The minimum distance the object can be from the connected anchor point.
Upper Translation	The maximum distance the object can be from the connected anchor point.
Break Force	Specify the linear (straight line) force level needed to break and so delete the joint. **Infinity** means it is unbreakable.
Break Torque	Specify the torque (rotation) level needed to break and so delete the joint. **Infinity** means it is unbreakable.

Details

(See also Joints 2D: *Details and Hints* for useful background information on all 2D joints.)

Use this joint to make objects slide! The aim of this joint is to maintain the position of two points on a configurable line that extends to infinity. Those two points can be two **Rigidbody2D** components or a **Rigidbody2D component** and a fixed position in the world. (Connect to a fixed position in the world by setting **Connected Rigidbody** to None).

The joint applies a linear force to both connected rigid body objects to keep them on the line. It also has a simulated linear motor that applies linear force to move the ridid body objects along the line. You can turn the motor off or on. Athough the line is infinite, you can specify just a segment of the line that you want to use, using the **Translation Limits** option.

This joint has three simultaneous constraints. All are optional:

Maintain a relative linear distance away from a specified line between two anchor points on two rigid body objects.
Maintain a linear speed between two anchor points on two rigid body objects along a specified line. (The speed is limited with a maximum force.)
Maintain a linear distance between two points along the specified line.

For Example:

You can use this joint to construct physical objects that need to react as if they are connected together on a line. Such as:

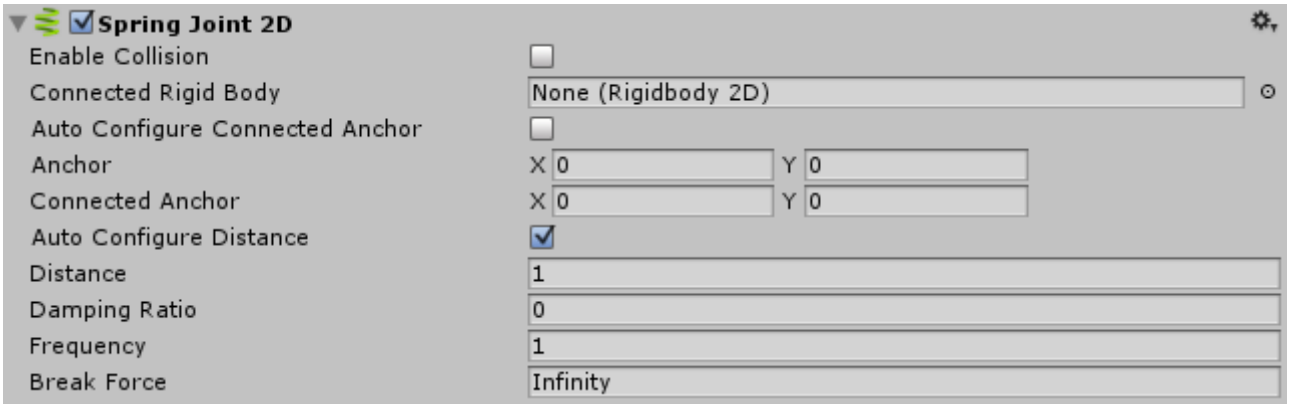
A platform which can move up or down. The platform reacts by moving down when something lands on it but must never move sideways. You can use this joint to ensure platform to never moves up or down beyond certain limits. Use the motor to make the platform move up.

Spring Joint 2D

Leave feedback

SWITCH TO SCRIPTING

The **Spring** Joint__ 2D__ **component** allows two game objects controlled by rigidbody physics to be attached together as if by a spring. The spring will apply a force along its axis between the two objects, attempting to keep them a certain distance apart.



Property:	*Function:*
Enable Collision	Can the two connected objects collide with each other? Check the box for yes.
Connected Rigid Body	Specify here the other object this joint connects to. Leave this as **None** and the other end of the joint will be fixed at a point in space defined by the **Connected Anchor** setting. Select the circle to the right of the field to view a list of objects to connect to.
Auto Configure Connected Anchor	Check this box to automatically set the anchor location for the other object this joint connects to. (Check this instead of completing the **Connected Anchor** fields.)
Anchor	The place (in terms of X, Y co-ordinates on the **RigidBody**) where the end point of the joint connects to *this* object.
Connected Anchor	The place (in terms of X, Y co-ordinates on the **RigidBody**) where the end point of the joint connects to *the other* object.
Auto Configure Distance	Check this box to automtically detect the distance between the two objects and set it as the distance that the joint keeps between the two objects.
Distance	The distance that the spring should attempt to maintain between the two objects. (Can be set manually.)
Damping Ratio	The degree to which you want to suppress spring oscillation: In the range 0 to 1, the higher the value, the less movement.
Frequency	The frequency at which the spring oscillates while the objects are approaching the separation distance you want (measured in cycles per second): In the range 0 to 1,000,000 - the higher the value, the stiffer the spring.
Break Force	Specify the force level needed to break and so delete the joint. **Infinity** means it is unbreakable.

Details

(See also Joints 2D: *Details and Hints* for useful background information on all 2D joints.)

This joint behaves like a spring. Its aim is to keep a linear distance between two points. You set this via the **Distance** setting. Those two points can be two **Rigidbody2D** components or a **Rigidbody2D** component and a fixed position in the world. (Connect to a fixed position in the world by setting **Connected Rigidbody** to None). The joint aplies a linear force to both rigid bodies. It doesn't apply torque (an angle force).

The joint uses a simulated spring. You can set the spring's stiffness and movement:

A stiff, barely moving spring...

A high (1,000,000 is the highest) **Frequency** == a stiff spring.

A high (1 is the highest) **Damping Ratio** == a barely moving spring.

A loose, moving spring...

A low **Frequency** == a loose spring.

A low **Damping Ratio** == a moving spring.

When the spring applies its force between the objects, it tends to overshoot the distance you have set between them, and then rebound repeatedly, giving in a continuous oscillation. The **Damping Ratio** sets how quickly the objects stop moving. The **Frequency** sets how quickly the objects oscillate either side of the target distance.

This joint has one constraint:

Maintain a zero linear distance between two anchor points on two rigid body objects.

For Example:

You can use this joint to construct physical objects that need to react as if they are connected together using a spring or a connection which allows rotation. Such as:

A character who's body is composed of multiple objects that act as if they are semi-rigid. Use the the **Spring Joint** to connect the character's body parts together, allowing them to flex to and from each other. You can specify whether the body parts are held together loosely or tightly.

HINTS:

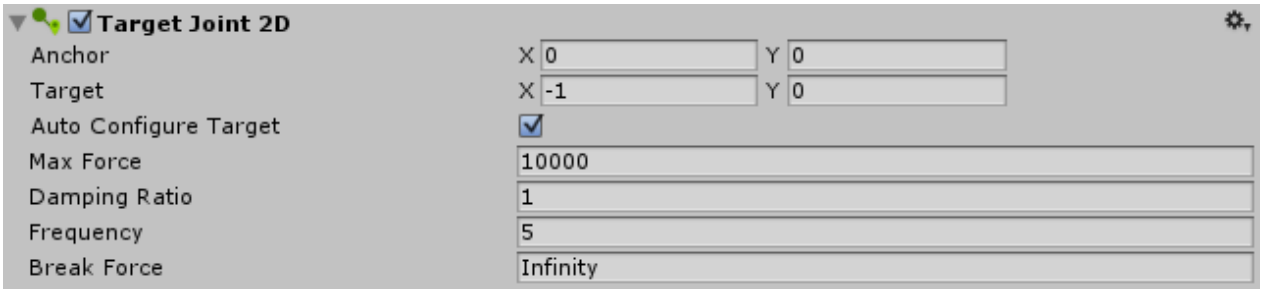
Zero in the **Frequency** is a special case: It gives the stiffest spring possible.
Spring Joint 2D uses a Box 2D spring-joint. Distance Joint 2D also uses the same Box 2D spring-joint but it sets the frequency to zero! Technically, a Spring Joint 2D with a frequency of zero and damping of 1 is identical to a Distance Joint 2D!

Target Joint 2D

Leave feedback

SWITCH TO SCRIPTING

This joint connects to a specified target, rather than another rigid body object, as other **joints** do. It is a spring type joint, which you could use for picking up and moving an object acting under gravity, for example.



Property:	*Function:*
Anchor	The place (in terms of X, Y co-ordinates on the **RigidBody**) where the end point of the joint connects to this object.
Target	The place (in terms of X, Y co-ordinates in the world space) towards which the other end of the joint attempts to move.
Auto Configure Target	Check this box to automatically set the other end of the joint to the current position of the object. (This option is useful when you are moving the object around, as it sets the initial target position as the current position.) **Note** that when this option is selected, the target changes as you move the object; it won't change if the option is unselected.
Max Force	Sets the force that the joint can apply when attempting to move the object to the **target position**. The higher the value, the higher the maximum force.
Damping Ratio	The degree to which you want to suppress spring oscillation: In the range 0 to 1, the higher the value, the less movement.
Frequency	The frequency at which the spring oscillates while the objects are approaching the separation distance you want (measured in cycles per second): In the range 0 to 1,000,000 -, the higher the value, the stiffer the spring.
Break Force	Specify the force level needed to break and so delete the joint. **Infinity** means it is unbreakable.

Details

(See also Joints 2D: *Details and Hints* for useful background information on all 2D joints.)

Use this joint to connect a rigid body game object to a point in space.

The aim of this joint is to keep zero linear distance between two points: An anchor point on a rigid body object and a world-space position, called the "**Target**". The joint applies linear force to the rigid body object, it does not apply torque (anglular force).

The joint uses a simulated spring. You can set the spring's stiffness and movement:

A stiff, barely moving spring...

A high (1,000,000 is the highest) **Frequency** == a stiff spring.

A high (1 is the highest) **Damping Ratio** == a barely moving spring.

A loose, moving spring...

A low **Frequency** == a loose spring.

A low **Damping Ratio** == a moving spring.

When the spring applies its force between the rigid body object and target, it tends to overshoot the distance you have set between them, and then rebound repeatedly, giving in a continuous oscillation. The **Damping Ratio** sets how quickly the rigid body object stops moving. The **Frequency** sets how quickly the rigid body object oscillates either side of the distance you have specified.

This joint has one constraint:

Maintain a zero linear distance between the anchor point on a rigid body object and a world-space position (**Target**).

For Example:

You can use this joint to construct physical objects that need to move to designated target positions and stay there until another target position is selected or the turned-off. Such as:

A game where players pick up cakes, using a mouse-click, and drag them into to a plate. You can use this joint to move each cake to the plate.

You could also use the joint to allow objects to hang: If the anchor point is not the **center of mass**, then the object will rotate. Such as:

A game where players pick up boxes. If they use a mouse-click to pick a box up by its corner and drag it, it will hang from the cursor.

HINT:

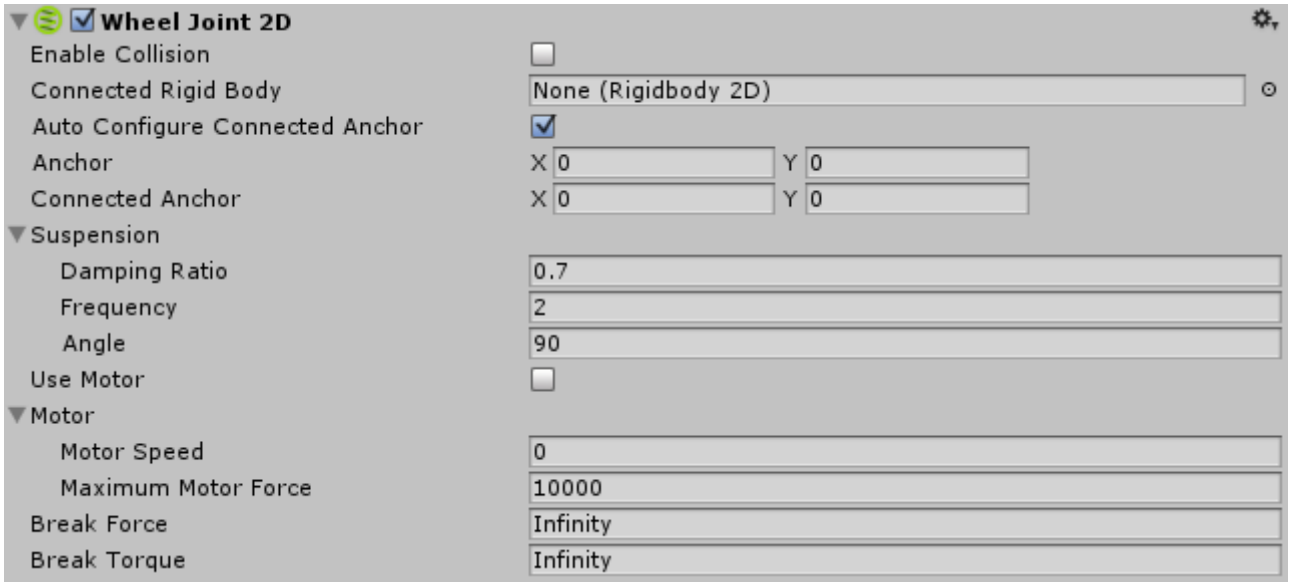
Zero in the **Frequency** is a special case: It gives the stiffest spring possible.

Wheel Joint 2D

Leave feedback

SWITCH TO SCRIPTING

Use the **Wheel** Joint__ 2D__ to simulate a rolling wheel, on which an object can move. You can apply motor power to the **joint**. The wheel uses a suspension "spring" to maintain its distance from the main body of the vehicle.



Properties

Property:	*Function:*
Enable Collision	Can the two connected objects collide with each other? Check the box for yes.
Connected Rigid Body	Specify here the other object this joint connects to. Leave this as **None** and the other end of the joint will be fixed at a point in space defined by the **Connected Anchor** setting. Select the circle to the right of the field to view a list of objects to connect to.
Auto Configure Connected Anchor	Check this box to automatically set the anchor location for the other object this joint connects to. (Check this instead of completing the **Connected Anchor** fields.)
Anchor	The place (in terms of X, Y co-ordinates on the **RigidBody**) where the end point of the joint connects to *this* object.
Connected Anchor	The place (in terms of X, Y co-ordinates on the **RigidBody**) where the end point of the joint connects to *the other* object.
Damping Ratio	The degree to which you want to suppress spring oscillation in the suspension: In the range 0 to 1, the higher the value, the less movement.
Frequency	The frequency at which the spring in the suspension oscillates while the objects are approaching the separation distance you want (measured in cycles per second): In the range 0 to 1,000,000 - the higher the value, the stiffer the suspension spring.
Angle	The world movement angle for the suspension.
Use Motor	Apply a motor force to the wheel? Check the box for yes.
Motor	

Property:	*Function:*
Motor Speed	Target speed (degrees per second) for the motor to reach.
Maximum Motor Force	Maximum force applied to the object to attain the desired speed.
Break Force	Specify the force level needed to break and so delete the joint. **Infinity** means it is unbreakable.
Break Torque	Specify the torque level needed to break and so delete the joint. **Infinity** means it is unbreakable.

Details

(See also Joints 2D: *Details and Hints* for useful background information on all 2D joints.)

Use this joint to simulate wheels and suspension. The aim of the joint is to keep the position of two points on a line that extends to infinity, whilst at the same time making them overlap. Those two points can be two **Rigidbody2D** components or a **Rigidbody2D component** and a fixed position in the world. (Connect to a fixed position in the world by setting **Connected Rigidbody** to None).

Wheel Joint 2D acts like a combination of a Slider Joint 2D (without its motor or limit constraints) and a Hinge Joint 2D (without its limit constraint).

The joint applies a linear force to both connected rigid body objects to keep them on the line, an angular motor to rotate the objects on the line, and a spring to simulate wheel suspension.

Set the **Maximum Motor Speed** and **Maximum Motor Force** (torque, in this joint) to control the angular motor speed, and make the two rigid body objects rotate.

You can set the wheel suspension stiffness and movement:

Stiff, barely moving suspension...

A high (1,000,000 is the highest) **Frequency** == stiff suspension.

A high (1 is the highest) **Damping Ratio** == barely moving suspension.

Loose, moving suspension...

A low **Frequency** == loose suspension.

A low **Damping Ratio** == moving suspension.

It has has two simultaneous constraints:

Maintain a zero relative linear distance away from a specified line between two anchor points on two rigid body objects.
Maintain an angular speed between two anchor points on two rigid body objects. (Set the speed via the **Maximum Motor Speed** option and maximum torque via **Maximum Motor Force**.)
For Example:

You can use this joint to construct physical objects that need to react as if they are connected with a rotational pivot but cannot move away from a specified line. Such as:

Simulating wheels with a motor to drive the wheels and a line defining the movement allowed for the suspension.

Hints

Wheel Joint 2D behaves differently to the Wheel Collider:

Unlike the Wheel Collider used with 3D physics, the Wheel Joint 2D uses a separate **Rigidbody** object for the wheel, which rotates when the force is applied. (The Wheel Collider, by contrast, simulates the suspension using a raycast and the wheel's rotation is purely a graphical effect). The wheel object will typically be a Circle Collider 2D with a Physics Material 2D that gives the right amount of traction for your gameplay.

To simulate a car or other vehicle:

Set the **Motor Speed** property to zero in the **inspector** and then vary it from your script according to the player's input. You can change the value of **Maximum Motor Force** to simulate the effect of gear changes and power-ups.

Zero frequency:

Zero in the **Frequency** is a special case: It gives the stiffest spring possible.

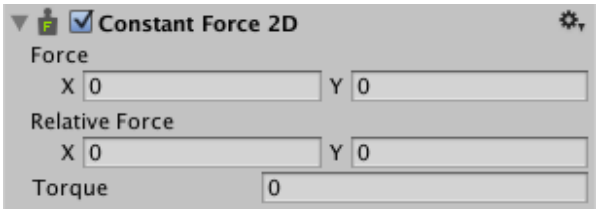
Constant Force 2D

Leave feedback

SWITCH TO SCRIPTING

Constant Force 2D is a quick utility for adding **constant forces** to a **Rigidbody 2D**. This works well for one-shot objects like rockets, if you want them to accelerate over time rather than starting with a large velocity.

Constant Force 2D applies both linear and torque (angular) forces continuously to the **Rigidbody** 2D, each time the **physics engine** updates at runtime.



The Constant Force 2D Inspector

Properties

Property:	*Function:*
Force	The linear force applied to the Rigidbody 2D at each physics update.
Relative Force	The linear force, relative to the Rigidbody 2D coordinate system, applied each physics update.
Torque	The torque applied to the Rigidbody 2D at each physics update.

Effectors 2D

Leave feedback

Use Effector 2D components with Collider 2D components to direct the forces of physics in your project when GameObject **colliders** come into contact with each other.

You can use Effector 2D components to create the following effects (among others):

Create 'platform' behaviour such as one-way **collisions** (PlatformEffector2D)
Create conveyor belts (SurfaceEffector2D)
Attract or repulse against a given source point (PointEffector2D)
Make GameOjects float (BuoyancyEffector2D)
Randomly vary force and angle magnitude (class-AreaEffector2D)

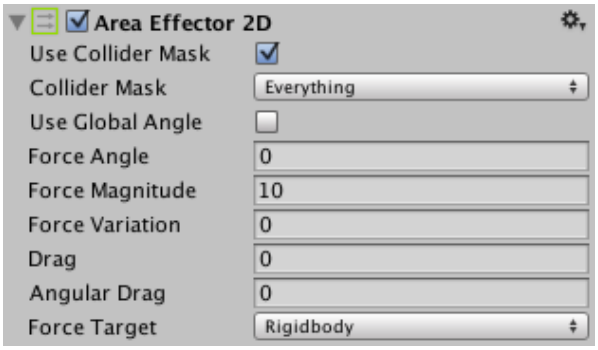
Area Effector 2D

Leave feedback

SWITCH TO SCRIPTING

The Area Effector 2D applies forces within an area defined by the attached **Collider** 2Ds when another (target) Collider 2D comes into contact with the Effector 2D. You can configure the force at any angle with a specific magnitude and random variation on that magnitude. You can also apply both linear and angular drag forces to slow down **Rigidbody** 2Ds.

Collider 2Ds that you use with the Area Effector 2D would typically be set as triggers, so that other Collider 2Ds can overlap with it to have forces applied. Non-triggers will still work, but forces will only be applied when Collider 2Ds come into contact with them.



The Area Effector 2D Inspector

Properties

Property:	*Function:*
Use Collider Mask	Check to enable use of the **Collider Mask** property? If this is not enabled, the Global **Collision** Matrix will be used as the default for all Collider 2Ds.
Collider Mask	The mask used to select specific Layers allowed to interact with the Area Effector 2D.
Use Global Angle	Check this to define the **Force Angle** as a global (world-space) angle. If this is not checked, the **Force Angle** is considered a local angle by the **physics engine**.
Force Angle	The angle of the force to be applied.
Force Magnitude	The magnitude of the force to be applied.
Force Variation	The variation of the magnitude of the force to be applied.
Drag	The linear drag to apply to Rigidbody 2Ds.
Angular Drag	The angular drag to apply to Rigidbody 2Ds.
Force Target	The point on a target **GameObject** where the Area Effector 2D applies any force.
Collider	The target point is defined as the current position of the Collider 2D. Applying force here can generate torque (rotation) if the Collider 2D isn't positioned at the **center of mass**.
Rigidbody	The target point is defined as the current center-of-mass of the Rigidbody 2D. Applying force here will never generate torque (rotation).

Buoyancy Effector 2D

Leave feedback

SWITCH TO SCRIPTING

The Buoyancy Effector 2D defines simple fluid behaviour such as floating and the drag and flow of fluid. You can also control a fluid surface, with the fluid behaviour taking place below.

The Buoyancy Effector 2D Inspector

Properties

Property:	*Function:*
Use Collider__An invisible shape that is used to handle physical collisions for an object. A collider doesn't need to be exactly the same shape as the object's mesh - a rough approximation is often more efficient and indistinguishable in gameplay. More info See in Glossary Mask__	Check this box to enable the '**Collider** Mask' property. If this is not enabled, the Global **Collision** Matrix will be used as the default for all **Collider** 2Ds.
Collider Mask	The mask used to select specific Layers allowed to interact with the effector. Note that this option only displays if you have selected **Use Collider Mask**.
Surface Level	Defines the surface location of the buoyancy fluid. When a **GameObject** is above this line, no buoyancy forces are applied. When a **GameObject** is intersecting or completely below this line, buoyancy forces are applied. This is a location specified as a world-space offset along the world y-axis, but is also scaled by the **GameObject**'s **Transform component**.
Density	The density of the Buoyancy Effector 2D fluid. This affects the behaviour of Collider 2Ds: Those with a higher density sink, those with a lower density float, and those with the same density appear suspended in the fluid.
Linear Drag	The drag coefficient affecting positional movement of a GameObject. This only applies when inside the fluid.
Angular Drag	The drag coefficient affecting rotational movement of a GameObject. This only applies when inside the fluid.

Property:	*Function:*
Flow Angle	The world-space angle (in degrees) for the direction of fluid flow. Fluid flow applies buoyancy forces in the specified direction.
Flow Magnitude	The “power” of the fluid flow force. Combined with **Fluid Angle**, this specifies the level of buoyancy force applied to GameObjects inside the fluid. The magnitude can also be negative, in which case the buoyancy forces are applied at 180 degrees to the **Flow Angle.**
Flow Variation	Enter a value here to randomly vary the fluid forces. Specify a positive or negative variation to randomly add or subtract from the **Fluid Magnitude**.

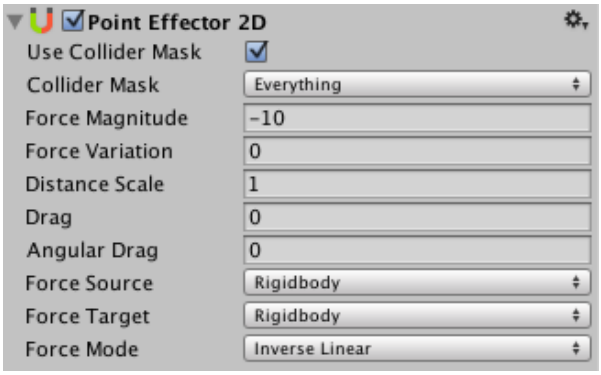
Point Effector 2D

Leave feedback

SWITCH TO SCRIPTING

The Point Effector 2D applies forces to attract/repulse against a source point which can be defined as the position of the rigid-body or the center of a collider used by the effector. When another (target) collider comes into contact with the effector then a force is applied to the target. Where the force is applied and how it is calculated can be controlled.

Colliders that you use with the effector would typically be set as triggers so that other colliders can overlap with it to have forces applied however, non-triggers will still work but forces will only be applied when colliders come into contact with it.



The Point Effector 2D Inspector

Properties

Property:	*Function:*
Use Collider Mask	Should the 'Collider Mask' property be used? If not then the global **collision** matrix will be used as is the default for all colliders.
Collider Mask	The mask used to select specific layers allowed to interact with the effector.
Force Magnitude	The magnitude of the force to be applied.
Force Variation	The variation of the magnitude of the force to be applied.
Distance Scale	The scale applied to the distance between the source and target. When calculating the distance, it is scaled by this amount allowing the effective distance to be changed which controls the magnitude of the force applied.
Drag	The linear drag to apply to rigid-bodies.
Angular Drag	The angular drag to apply to rigid-bodies.
Force Source	The force source is the point that attracts or repels target objects. The distance from the target is defined from this point.
Collider	The source point is defined as the current position of the collider.
Rigidbody	The source point is defined as the current position of the rigidbody.
Force Target	The force target is the point on a target object where the effector applies any force. The distance to the source is defined from this point.
Collider	The target point is defined as the current position of the collider. Applying force here can generate torque (cause the target to rotate) if the collider isn't positioned at the center-of-mass.
Rigidbody	The target point is defined as the current center-of-mass of the rigidbody. Applying force here will never generate torque (cause the target to rotate).
Force Mode	How the force is calculated.
Constant	The force is applied ignoring how far apart the source and target are.

Property:	*Function:*
Inverse Linear	The force is applied as a function of the inverse-linear distance between the source and target. When the source and target are in the same position then the full force is applied but it falls-off linearly as they move apart.
Inverse Squared	The force is applied as a function of the inverse-squred distance between the source and target. When the source and target are in the same position then the full force is applied but it falls-off squared as they move apart. This is similar to real-world gravity.

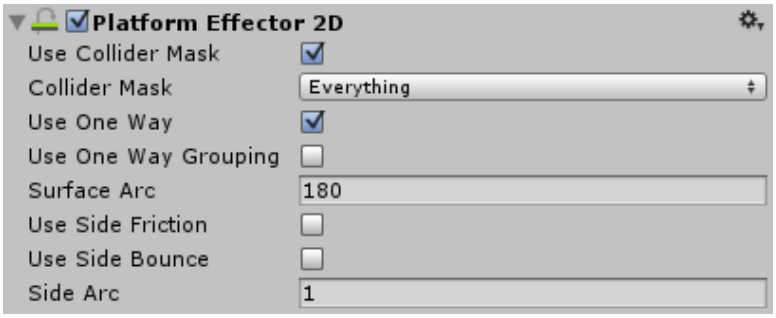
Platform Effector 2D

Leave feedback

SWITCH TO SCRIPTING

The Platform Effector 2D applies various “platform” behaviour such as one-way **collisions**, removal of side-friction/bounce **etc**.

Colliders that you use with the effector would typically not be set as triggers so that other **colliders** can collide with it.



The Platform Effector 2D Inspector

Properties

Property:	*Function:*
Use Collider Mask	Should the ‘Collider Mask’ property be used? If not then the global collision matrix will be used as is the default for all colliders.
Collider Mask	The mask used to select specific layers allowed to interact with the effector.
Use One Way	Should one-way collision behaviour be used?
Use One Way Grouping	Ensures that all contacts disabled by the one-way behaviour act on all colliders. This is useful when multiple colliders are used on the object passing through the platform and they all need to act together as a group.
Surface Arc	The angle of an arc centered on the local ‘up’ the defines the surface which doesn’t allow colliders to pass. Anything outside of this arc is considered for one-way collision.
Use Side Friction	Should friction should be used on the platform sides?
Use Side Bounce	Should bounce should be used on the platform sides?
Side Arc	The angle of an arc that defines the sides of the platform centered on the local ‘left’ and ‘right’ of the effector. Any collision normals within this arc are considered for the ‘side’ behaviours.

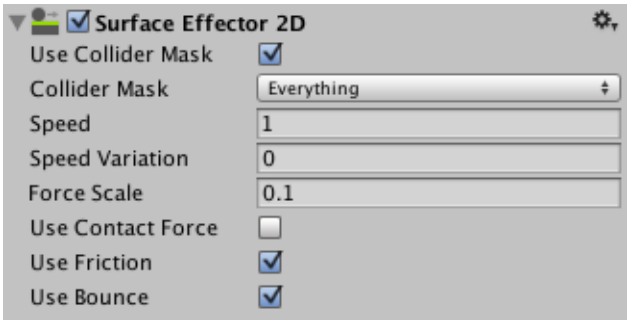
Surface Effector 2D

Leave feedback

SWITCH TO SCRIPTING

The Surface Effector 2D applies tangent forces along the surfaces of **colliders** used by the effector in an attempt to match a specified speed along the surface. This is analogous to a conveyor belt.

Colliders that you use with the effector would typically be set as non-triggers so that other colliders can come into contact with the surface.



The Surface Effector 2D Inspector

Properties

Property:	*Function:*
Use Collider Mask	Should the 'Collider Mask' property be used? If not then the global **collision** matrix will be used as is the default for all colliders.
Collider Mask	The mask used to select specific layers allowed to interact with the effector.
Speed	The speed to be maintained along the surface.
Speed Variation	A random increase in the speed (anywhere between 0 and the Speed Variation value) will be applied to the speed. If a negative number is entered here, a random *reduction* in the speed will be applied.
Force Scale	Allows scaling of the force that is applied when the effector attempts to attain the specified speed along the surface. If 0 then no force is applied is is effectively disabled. If 1 then the full force is applied. It can be thought of as how fast the target object is modified to meet the specified speed with lower values being slower and higher values being quicker. Care should be taken however as applying the full force will easily counteract any other forces being applied to the target object such as jump or other movement forces. For this reason, a value less than 1 is advised.
Use Contact Force	Should the force be applied at the point of contact between the surface and the target collider? Using contact forces can cause the target object to rotate whereas not using contact forces won't.
Use Friction	Should friction be used when contacting the surface?
Use Bounce	Should bounce be used when contacting the surface?

Graphics

Leave feedback

Make your game look just how you envisaged it with Real-time Global Illumination and our physically-based **shader**. From luminous day, to the gaudy glow of neon signs at night; from sunshafts, to dimly lit midnight streets and shadowy tunnels – create an evocative dynamic game to enthrall players on any platform.

This section explains all you need to know about Lighting, Cameras, Materials, Shaders & Textures, Particles & Visual Effects, and much more.

See also the Knowledge Base Graphics section.

There are also many useful graphics tutorials in the Tutorials section.

Related tutorials: Graphics

See the Knowledge Base Editor section for troubleshooting, tips and tricks.

Graphics Overview

Leave feedback

Understanding graphics is key to adding an element of depth to your game. This section covers the graphical features of Unity, such as lighting and **rendering**.

Lighting

Leave feedback

This section details the advanced lighting features available in Unity.

For an introduction, see the Lights manual page and the Light component reference page.

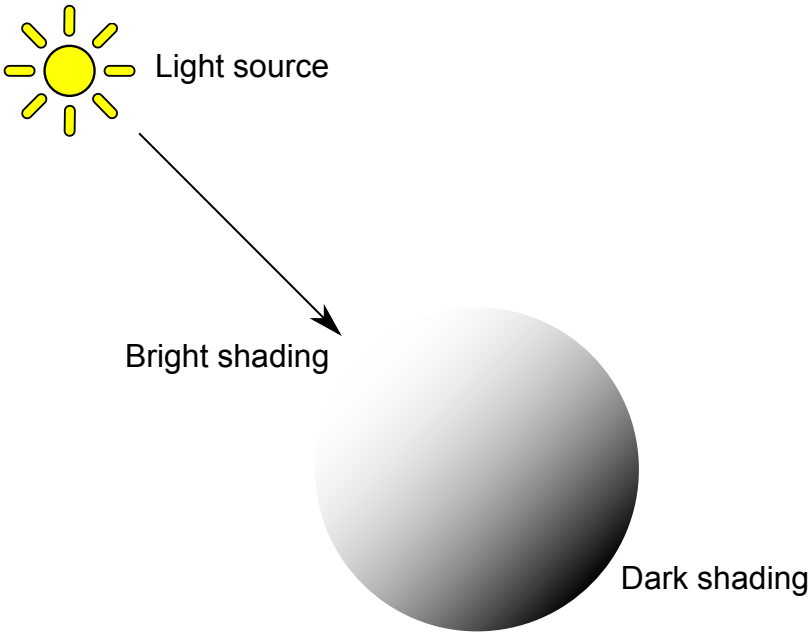
There are also lighting tutorials in the Tutorials section.

For specific problems, try searching the Knowledge Base, or asking on the Forums.

Lighting overview

Leave feedback

In order to calculate the shading of a **3D object**, Unity needs to know the intensity, direction and color of the light that falls on it.



These properties are provided by `Light` objects in the scene. Different types of light emit their assigned color in different ways; some lights may diminish with distance from the source, and have different rules about the angle of light received from the source. The different types of light source available in Unity are detailed in Types of light.

Unity can calculate complex, advanced lighting effects in various different ways, each suited to different use cases.

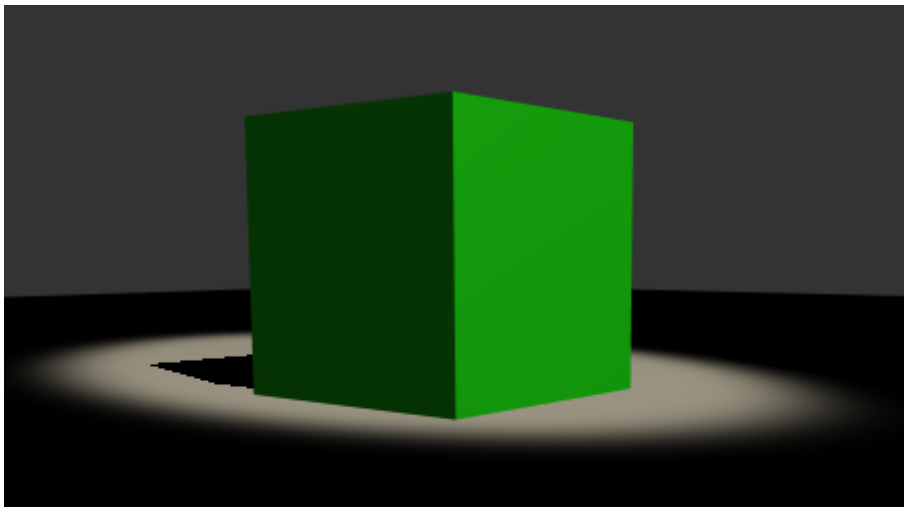
Choosing a lighting technique

Broadly speaking, lighting in Unity can be considered as either 'realtime' or 'precomputed' in some way and both techniques can be used in combination to create immersive **scene** lighting.

In this section we will give a brief overview of what opportunities the different techniques offer, their relative advantages and individual performance characteristics.

Realtime lighting

By default, lights in Unity - directional, spot and point, are realtime. This means that they contribute direct light to the scene and update every frame. As lights and **GameObjects** are moved within the scene, lighting will be updated immediately. This can be observed in both the scene and game views.



The effect of realtime light alone. Note that shadows are completely black as there is no bounced light. Only surfaces falling within the cone of the spot light are affected.

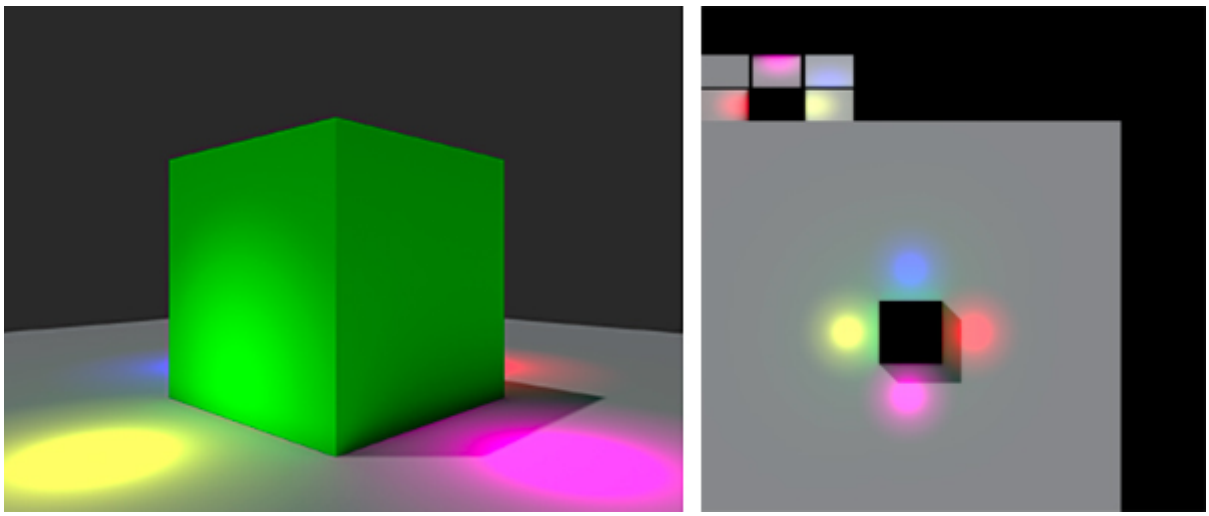
Realtime lighting is the most basic way of lighting objects within the scene and is useful for illuminating characters or other movable geometry.

Unfortunately, the light rays from Unity's realtime lights do not bounce when they are used by themselves. In order to create more realistic scenes using techniques such as global illumination we need to enable Unity's precomputed lighting solutions.

Baked lightmaps

Unity can calculate complex static lighting effects (using a technique called global illumination, or GI) and store them in a reference texture map called a **lightmap**. This calculation process is referred to as baking.

When baking a lightmap, the effects of light sources on static objects in the scene are calculated and the results are written to textures which are overlaid on top of scene geometry to create the effect of lighting.



Left: A simple lightmapped scene. Right: The lightmap texture generated by Unity. Note how both shadow and light information is captured.

These lightmaps can include both the direct light which strikes a surface and also the indirect light that bounces from other objects or surfaces within the scene. This lighting texture can be used together with surface information like color (albedo) and relief (normals) by the **Shader** associated with an object's material.

With baked lighting, these lightmaps cannot change during gameplay and so are referred to as 'static'. Realtime lights can be overlaid and used additively on top of a lightmapped scene but cannot interactively change the lightmaps themselves.

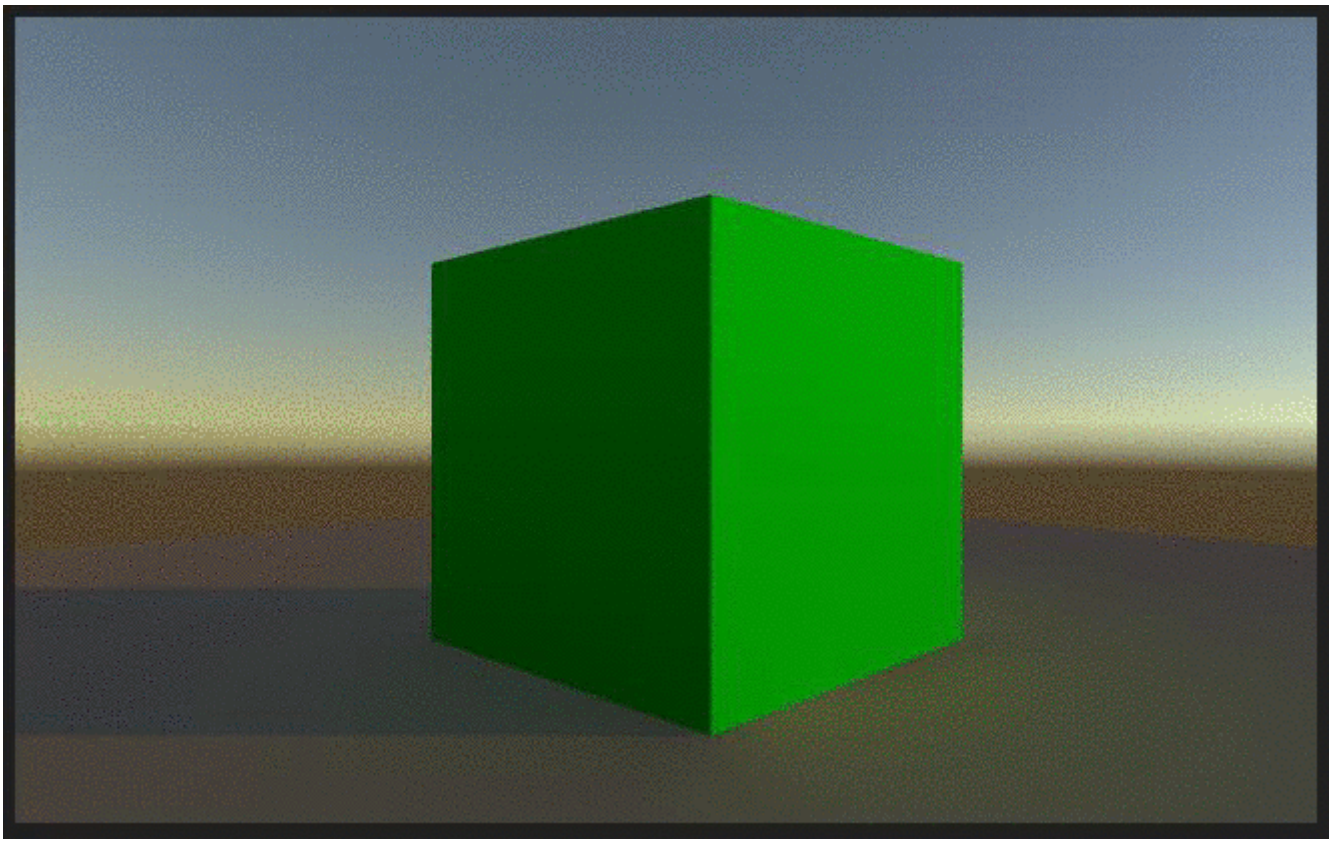
With this approach, we trade the ability to move our lights at gameplay for a potential increase in performance, suiting less powerful hardware such as mobile platforms.

See the Lighting window reference and Using precomputed lighting for more information.

Precomputed realtime global illumination

Whilst static lightmaps are unable to react to changes in lighting conditions within the scene, precomputed realtime GI does offer us a technique for updating complex scene lighting interactively.

With this approach it is possible to create lit environments featuring rich global illumination with bounced light which responds, in realtime, to lighting changes. A good example of this would be a time of day system - where the position and color of the light source changes over time. With traditional baked lighting, this is not possible.



A simple example of time of day using Precomputed Realtime GI.

In order to deliver these effects at playable framerates, we need to shift some of the lengthy number-crunching from being a realtime process to one which is precomputed.

Precomputing shifts the burden of calculating complex light behaviour from something that happens during gameplay, to something which can be calculated when time is no longer so critical. We refer to this as an 'offline' process.

For further information, please see the lighting and rendering tutorial.

Benefits and costs

Although it is possible to simultaneously use Baked GI lighting and Precomputed Realtime GI, be wary that the performance cost of **rendering** both systems simultaneously is exactly the sum of them both. Not only do we have to store both sets of lightmaps in video memory, but we also pay the processing cost of decoding both in shaders.

The cases in which you may wish to choose one lighting method over another depend on the nature of your project and the performance capabilities of your intended hardware. For example, on mobile where video memory and processing power is more limited, it is likely that a Baked GI lighting approach would be more performant. On standalone computers with dedicated graphics hardware, or recent games consoles, it is quite possible to use Precomputed Realtime GI or even to use both systems simultaneously.

The decision on which approach to take will have to be evaluated based on the nature of your particular project and desired target platform. Remember that when targeting a range of different hardware, that often it is the least performant which will determine which approach is needed.

See also: Light Troubleshooting and Performance

Lighting Window

Leave feedback

The Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**) is the main control point for Unity's Global Illumination (GI) features. Although GI in Unity gives good results with default settings, the Lighting window's properties allow you to adjust many aspects of the GI process, to customise your Scene or optimise for quality, speed and storage space as you need. This window also includes settings for ambient light, halos, cookies and fog.

Overview

The controls of the Lighting window are divided among three tabs:

The **Scene** tab settings apply to the overall Scene rather than individual **GameObjects**. These settings control lighting effects and also optimisation options.

The **Global maps** tab shows all of the **lightmap Asset** files generated by the GI lighting process.

The **Object maps** tab shows previews of GI lightmap textures (including shadow masks) for the currently selected GameObject.

The window also has an **Auto Generate** checkbox below the displayed content. If you tick this checkbox, Unity updates lightmap data as you edit the Scene. Note that the update usually takes a few seconds rather than happening instantaneously. If you leave the **Auto Generate box** unticked, the **Generate Lighting** button to the right of the checkbox becomes active; use this button to trigger lightmap updates when you need them. Use the **Generate Lighting** button if you want to clear the baked data from the Scene without clearing the GI Cache.

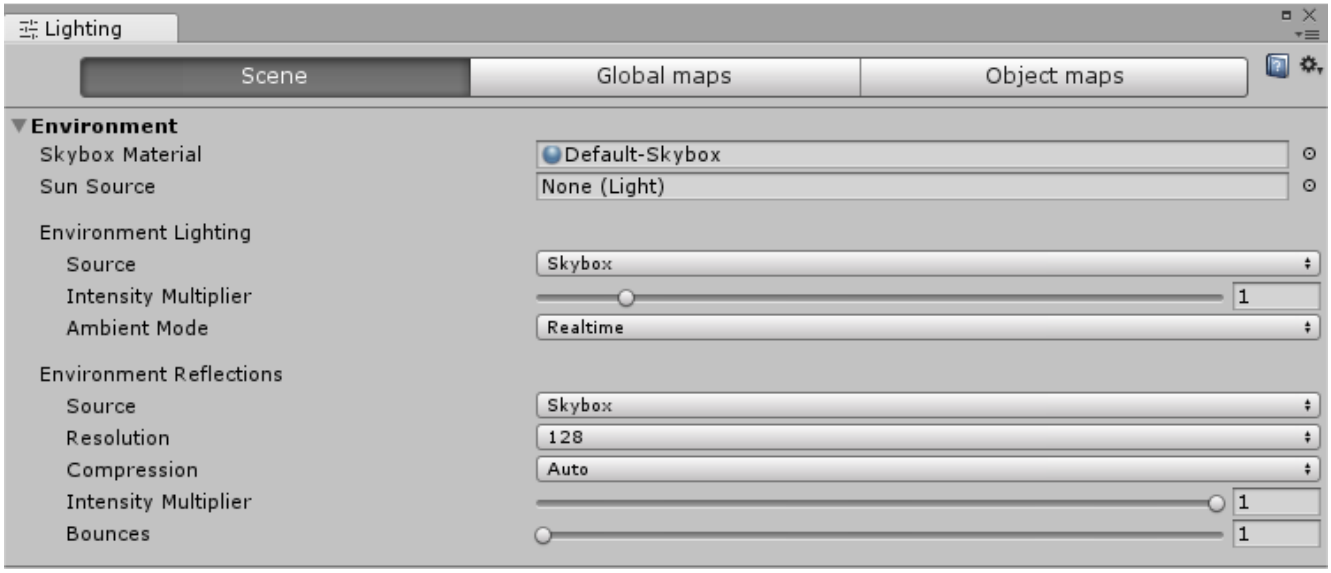
Scene tab

The **Scene** tab contains settings that apply to the overall Scene, rather than individual GameObjects. The **Scene** tab contains several sections:

Environment
Realtime Lighting
Mixed Lighting
Lightmapping Settings
Other Settings
Debug Settings

Environment

The Environmental Lighting section contains settings for the skybox, diffuse lighting and reflections.

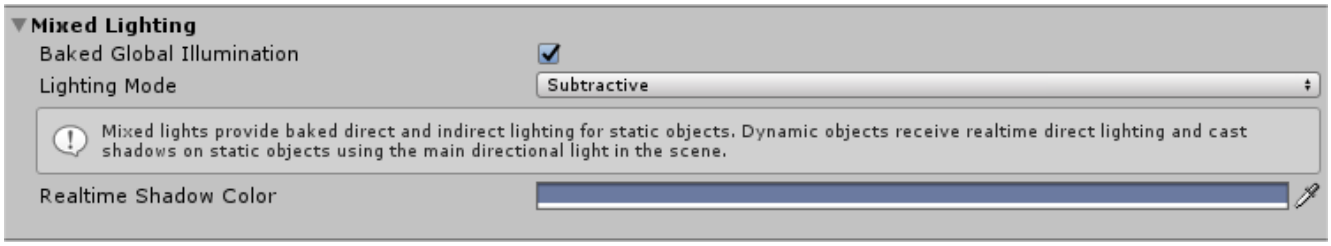


Property:	Function:
Skybox Material	A skybox is a Material that appears behind everything else in the Scene to simulate the sky or other distant background. Use this property to choose the skybox Material you want to use for the Scene. The default value is the Material **Default-Skybox** in the Standard Assets.
Sun Source	When a procedural skybox is used, use this to specify a GameObject with a directional Light **component** to indicate the direction of the "sun" (or whatever large, distant light source is illuminating your Scene). If this is set to **None** (the default), the brightest directional light in the Scene is assumed to represent the sun.
Environment Lighting	These settings affect light coming from the distant environment.
Source	Diffuse environmental light (also known as ambient light) is light that is present all around the Scene and doesn't come from any specific source object. Use this to define a source colour. The default value is **Skybox**.
Color	Select this to use a flat color for all ambient light in the Scene.
Gradient	Select this to choose separate colors for ambient light from the sky, horizon and ground, and blend smoothly between them.
Skybox	Select this to use the colors of the skybox (if specified by the **Skybox Material**) to determine the ambient light coming from different angles. This allows for more precise effects than **Gradient**.
Intensity Multiplier	Use this to set the brightness of the diffuse environmental light in the Scene, defined as a value between 0 and 8. The default value is 1.
Ambient Mode	Use this to specify the Global Illumination mode that should be used to handling ambient light in the Scene. This property is only available when both real-time lighting and baked lighting are enabled in the Scene.
Realtime	Choose **Realtime** if you want the ambient light in the Scene to be calculated and updated in real time.
Baked	Choose **Baked** if you want the ambient light to be precomputed and set into the Scene at run time.
Environment Reflections	These settings control global settings involved in Reflection Probe baking, and settings affecting global reflections.
Source	Use this setting to specify whether you want to use the skybox for reflection effects, or a cube map of your choice. The default value is **Skybox**.
Skybox	Select this to use the skybox for reflections. If you select **Skybox**, an additional option called **Resolution** appears. Use this to set the resolution of the skybox for reflection purposes.
Custom	Select this to use a cube map for reflections. If you select Custom, an additional option called **Cubemap** appears. Use this to set the cube map of the skybox for reflection purposes.
Compression	Use this to define whether or not reflection textures are compressed. The default setting is **Auto**.
Auto	The reflection texture is compressed if the compression format is suitable.
Uncompressed	The reflection texture is stored in memory uncompressed.
Compressed	The texture is compressed.
Intensity Multiplier	The degree to which the reflection source (the skybox or cube map specified in the **Reflection Source** property) is visible in reflective objects.
Bounces	A reflection "bounce" occurs when a reflection from one object is then reflected by another object. The reflections are captured in the scene through the use of Reflection Probes. Use this property to set how many times the Reflection Probes evaluate bounces back and forth between objects. If this is set to 1, then Unity only takes the initial reflection (from the skybox or cube map specified in the **Reflection Source** property) into account.

Realtime Lighting

Property:	Function:
Realtime Global Illumination	If this checkbox is ticked, Unity calculates and updates the lighting in real time. See documentation on Realtime Global Illumination for more information.

Mixed Lighting



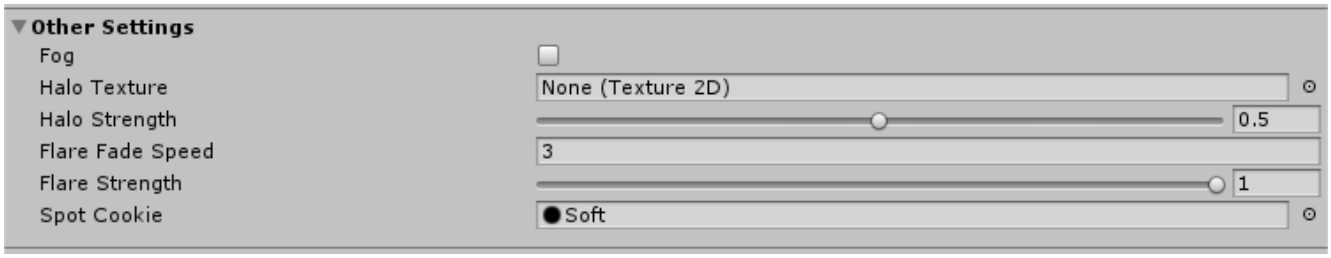
Property:	Function:
Baked Global Illumination	If this checkbox is ticked, Unity precomputes the lighting and sets it into the Scene at run time. See documentation on Baked Global Illumination for more information.
Lighting Mode	**Lighting Mode** determines the way Mixed Lights and shadows work with GameObjects in the Scene. **Note**: When you change the **Lighting Mode**, you also need to re-bake the Scene. If **Auto Generate** is enabled in the Lighting window, this happens automatically. If **Auto Generate** is not enabled, click **Generate Lighting** to see the updated lighting effect.
Realtime Shadow Color	Define the color used to render real-time shadows. This setting is only available when **Lighting Mode** is set to **Subtractive**.

Lightmapping Settings

Lightmapping Settings are specific to the **Lightmapper** backend. Unlike other settings, they're not shared and have their own parameters. To see full settings information for each lighting system, see documentation on Progressive Lightmapper and Enlighten.

Other Settings

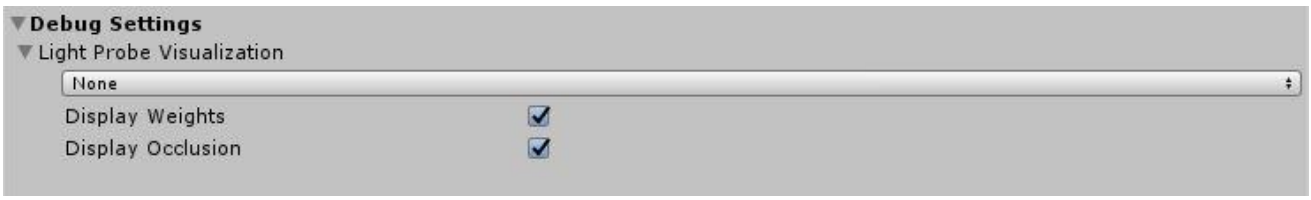
This section provides documentation for the **Other Settings** section.



Property:	Function:
Other Settings	Settings for fog, Halos, Flares and Cookies.
Fog	Enables or disables fog in the Scene. Note that fog is not available with the Deferred rendering path. For deferred rendering, the Post-processing fog effect effect might be suitable.
Color	Use the color picker to set the color Unity uses to draw fog in the Scene.
Mode	Define the way in which the fogging accumulates with distance from the **camera**.
Linear	Fog density increases linearly with distance.
Start	Set the distance from the Camera at which the fog starts.

Property:	Function:
End	Set the distance from the Camera at which the fog completely obscures Scene GameObjects.
Exponential	Fog density increases exponentially with distance.
Density	Use this to control the density of the fog. The fog appears more dense as the **Density** increases.
Exponential Squared	Fog density increases faster with distance (exponentially and squared).
Density	Use this to control the density of the fog. The fog appears more dense as the **Density** increases.
Halo Texture	Set the Texture you want to use for drawing a Halo around lights.
Halo Strength	Define the visibility of Halos around Lights, from a value between 0 and 1.
Flare Fade Speed	Define the time (in seconds) over which lens flares fade from view after initially appearing. This is set to 3 by default.
Flare Strength	Define the visibility of lens flares from lights, from a value between 0 and 1.
Spot Cookie	Set the Cookie texture you want to use for spot lights.

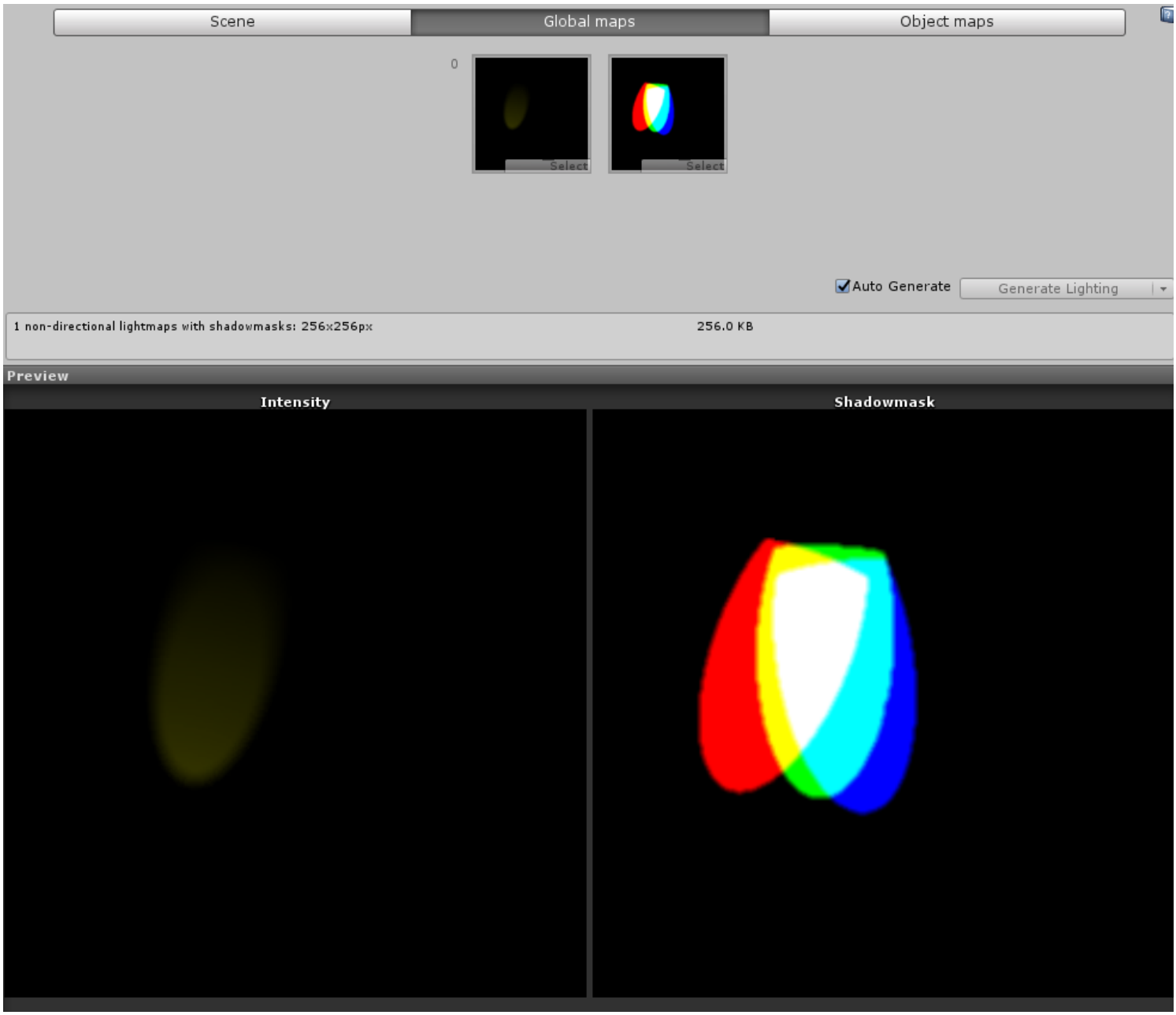
Debug settings



Property:	**Function:**
Debug Settings	Settings that help you debug your Scene.
Light Probe Visualization	Use this to filter which Light Probes are visualized in the **Scene view**. The default value is **Only Probes Used By Selection**.
Only Probes Used By Selection	Only Light Probes that affect the current selection are visualized in the Scene view.
All Probes No Cells	All Light Probes are visualized in the Scene view.
All Probes With Cells	All Light Probes are visualized in the Scene view, and the tetrahedrons that are used for **interpolation** of Light Probe data are also displayed.
None	No Light Probes are visualized in the Scene view.
Display Weights	When ticked, Unity draws a line from the Light Probe used for the active selection to the positions on the tetrahedra used for interpolation. This is a way to debug probe interpolation and placement problems.
Display Occlusion	When ticked, Unity displays occlusion data for Light Probes if the Mixed lighting mode is Distance Shadowmask or Shadowmask.

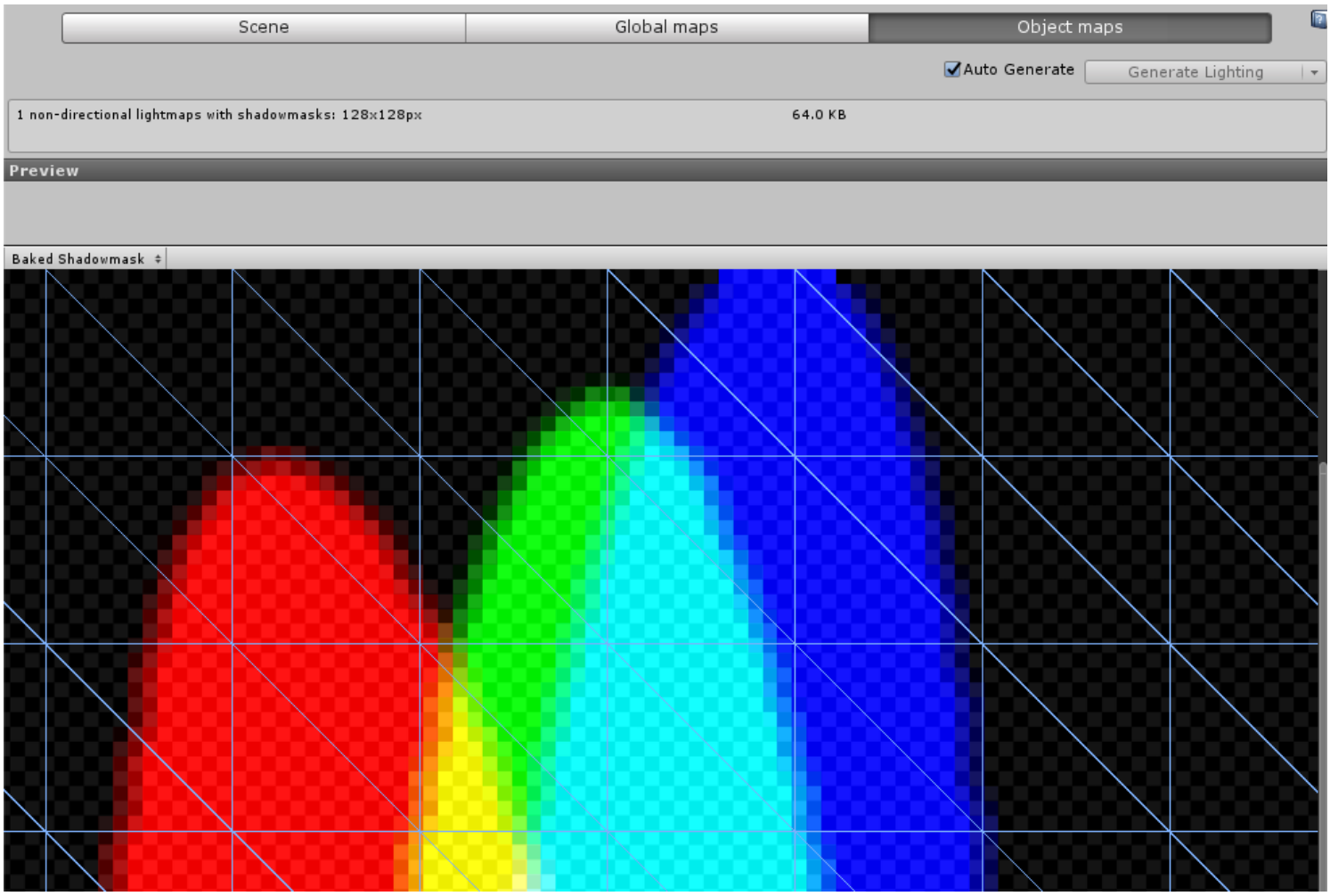
Global maps tab

Use the **Global maps tab** to view the actual textures in use by the lighting system. These include intensity light maps, shadow masks and directionality maps. This is only available when Baked lighting or Mixed lighting is used; the preview is blank for Realtime lighting.



Object maps tab

Use the **Object maps** tab to see previews of baked textures for the currently selected* *GameObject only, including shadow masks.



2017–06–08 Page published with limited editorial review

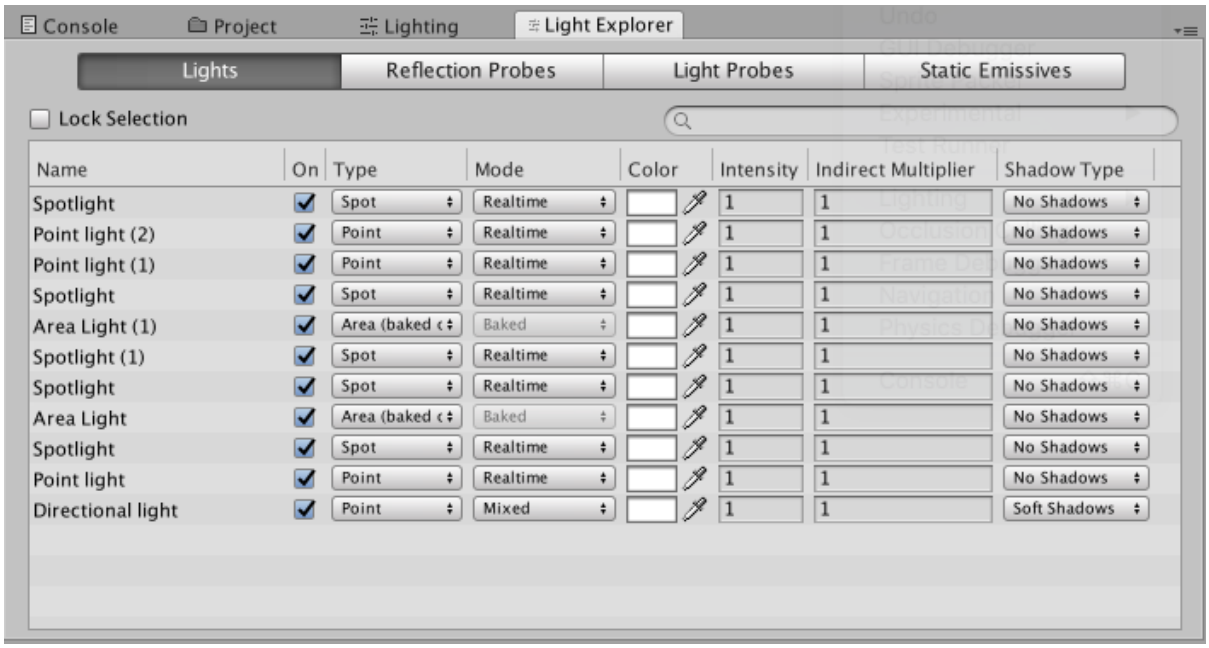
Updated in 5.6

Lightmapping section and Debug settings updated in 2018.1

Light Explorer

Leave feedback

The Light Explorer allows you to select and edit light sources. To open the Light Explorer from the menu, navigate to **Window** > **Rendering** > **Light Explorer**.



Use the four tabs at the top of the panel to view settings for the Lights, Reflection Probes, Light Probes, and Static Emissives in the current **Scene**. The editable parameters are the most commonly-used fields for each **component** type.

Use the search field to filter each table for names. You can also select the lights you want to work on, then tick the **Lock Selection** checkbox. Only the lights that were selected when the checkbox was ticked remain listed in the Light Explorer, even if you select a different Light in the Scene.

2017–06–08 Page published with limited editorial review

Updated in 5.6

Light sources

Leave feedback

Lighting in Unity is primarily provided by Light objects. There are also two other ways of creating light (ambient light and emissive materials), depending on the method of lighting you have chosen.

The following sections detail the various ways of creating light in Unity.

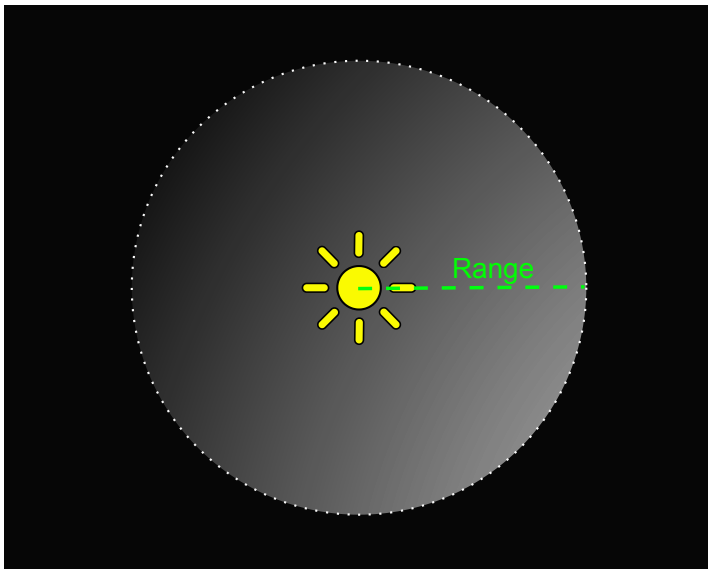
Types of light

Leave feedback

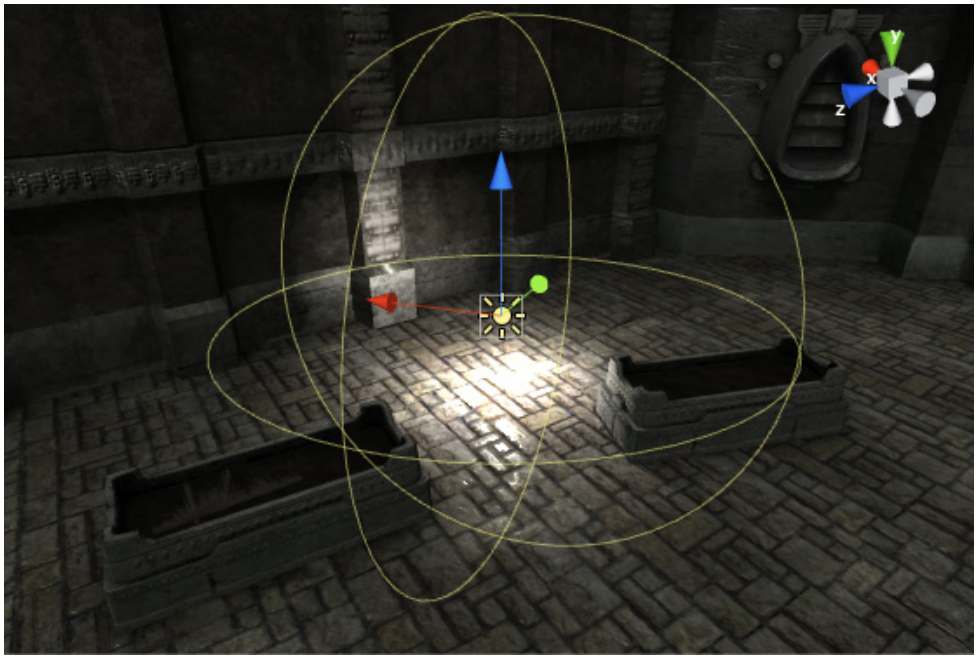
This section details the many different ways of creating light in Unity.

Point lights

A point light is located at a point in space and sends light out in all directions equally. The direction of light hitting a surface is the line from the point of contact back to the center of the light object. The intensity diminishes with distance from the light, reaching zero at a specified range. Light intensity is inversely proportional to the square of the distance from the source. This is known as 'inverse square law' and is similar to how light behaves in the real world.



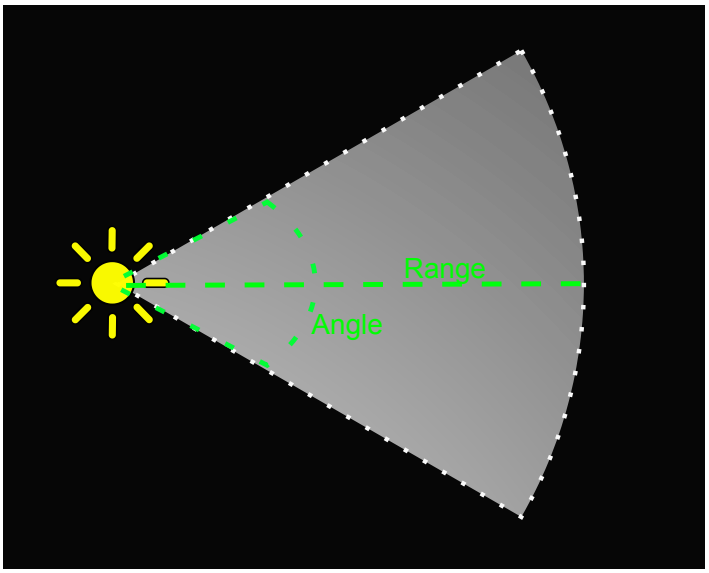
Point lights are useful for simulating lamps and other local sources of light in a **scene**. You can also use them to make a spark or explosion illuminate its surroundings in a convincing way.



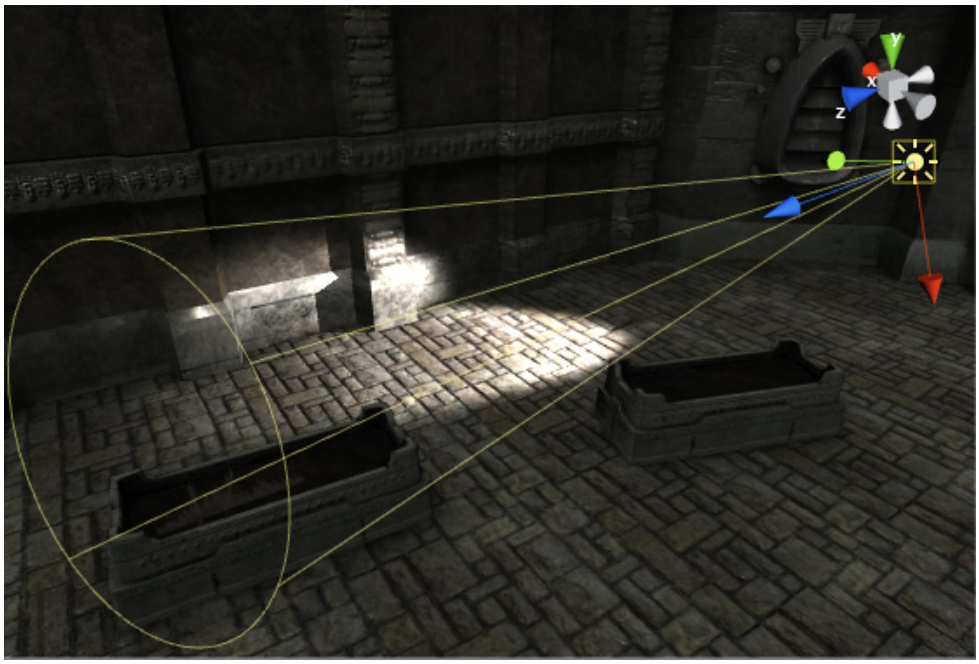
Effect of a Point Light in the scene

Spot lights

Like a point light, a spot light has a specified location and range over which the light falls off. However, the spot light is constrained to an angle, resulting in a cone-shaped region of illumination. The center of the cone points in the forward (Z) direction of the light object. Light also diminishes at the edges of the spot light's cone. Widening the angle increases the width of the cone and with it increases the size of this fade, known as the 'penumbra'.



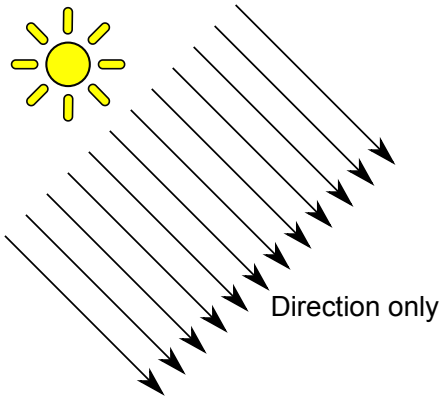
Spot lights are generally used for artificial light sources such as flashlights, car headlights and searchlights. With the direction controlled from a script or animation, a moving spot light will illuminate just a small area of the scene and create dramatic lighting effects.



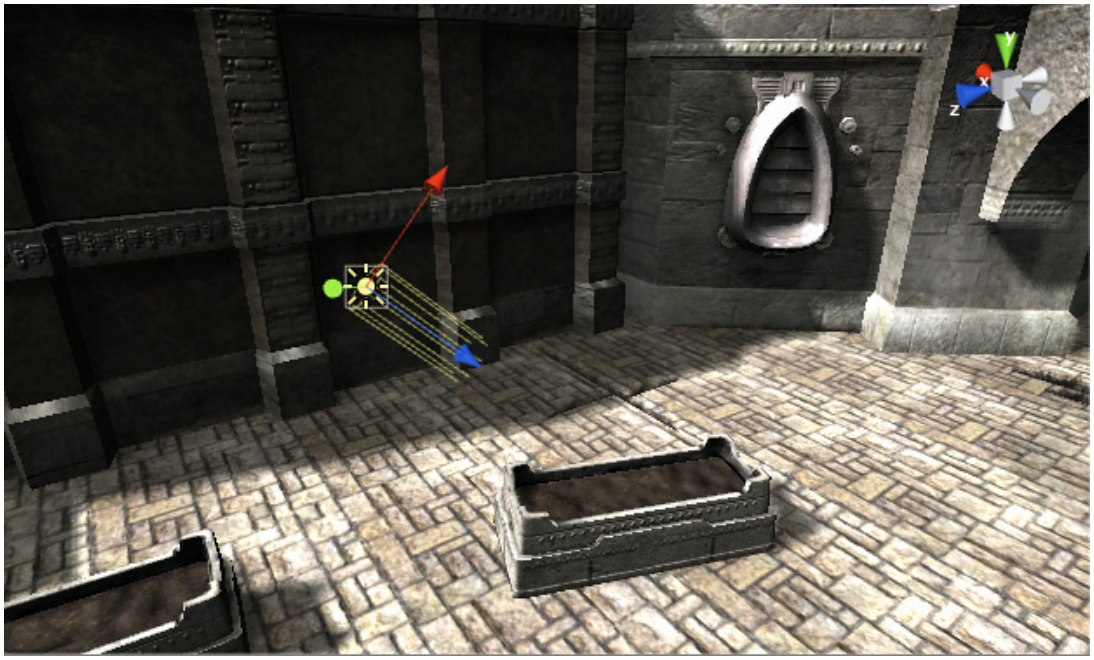
Effect of a Spot Light in the scene

Directional lights

Directional lights are very useful for creating effects such as sunlight in your scenes. Behaving in many ways like the sun, directional lights can be thought of as distant light sources which exist infinitely far away. A directional light does not have any identifiable source position and so the light object can be placed anywhere in the scene. All objects in the scene are illuminated as if the light is always from the same direction. The distance of the light from the target object is not defined and so the light does not diminish.



Directional lights represent large, distant sources that come from a position outside the range of the game world. In a realistic scene, they can be used to simulate the sun or moon. In an abstract game world, they can be a useful way to add convincing shading to objects without exactly specifying where the light is coming from.

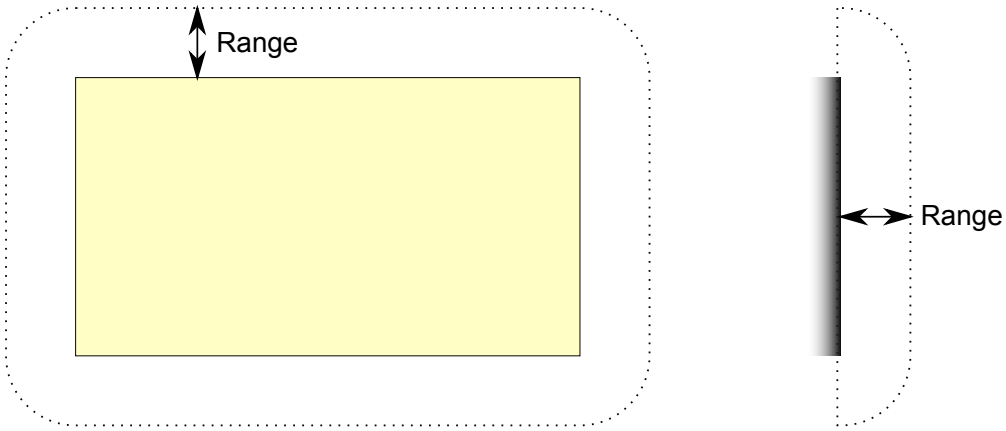


Effect of a Directional Light in the scene

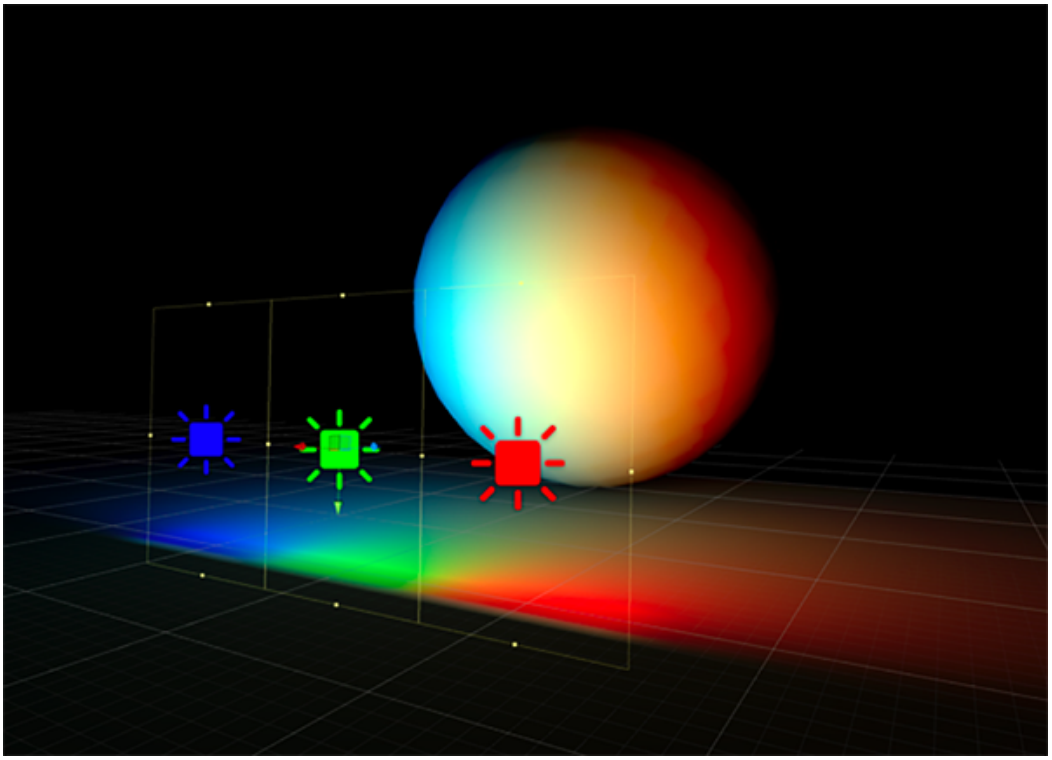
By default, every new Unity scene contains a Directional Light. In Unity 5, this is linked to the procedural sky system defined in the Environment Lighting section of the Lighting Panel (Lighting>Scene>Skybox). You can change this behaviour by deleting the default Directional Light and creating a new light or simply by specifying a different GameObject from the 'Sun' parameter (Lighting>Scene>Sun). Rotating the default Directional Light (or 'Sun') causes the '**Skybox**' to update. With the light angled to the side, parallel to the ground, sunset effects can be achieved. Additionally, pointing the light upwards causes the sky to turn black, as if it were nighttime. With the light angled from above, the sky will resemble daylight. If the Skybox is selected as the ambient source, Ambient Lighting will change in relation to these colors.

Area lights

An `Area Light` is defined by a rectangle in space. Light is emitted in all directions uniformly across their surface area, but only from one side of the rectangle. There is no manual control for the range of an Area Light, however intensity will diminish at inverse square of the distance as it travels away from the source. Since the lighting calculation is quite processor-intensive, area lights are not available at runtime and can only be baked into lightmaps.

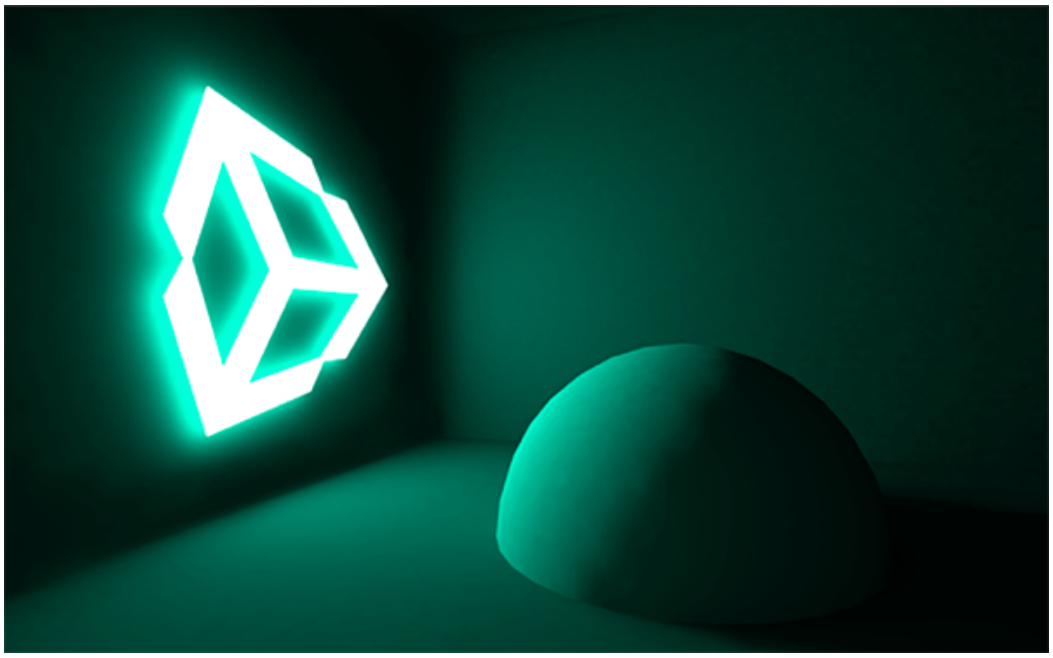


Since an area light illuminates an object from several different directions at once, the shading tends to be more soft and subtle than the other light types. You might use it to create a realistic street light or a bank of lights close to the player. A small area light can simulate smaller sources of light (such as interior house lighting) but with a more realistic effect than a point light.



Light is emitted across the surface of an Area Light producing a diffuse light with soft shadowing.

Emissive materials



Like area lights, emissive materials emit light across their surface area. They contribute to bounced light in your scene and associated properties such as color and intensity can be changed during gameplay. Whilst area lights are not supported by Precomputed Realtime GI, similar soft lighting effects in realtime are still possible using emissive materials.

'Emission' is a property of the **Standard Shader** which allows static objects in our scene to emit light. By default the value of 'Emission' is set to zero. This means no light will be emitted by objects assigned materials using the **Standard Shader**.

There is no range value for emissive materials but light emitted will again falloff at a quadratic rate. Emission will only be received by objects marked as 'Static' or "**Lightmap** Static' from the **Inspector**. Similarly, emissive materials applied to non-static, or dynamic geometry such as characters will not contribute to scene lighting.

However, materials with an emission above zero will still appear to glow brightly on-screen even if they are not contributing to scene lighting. This effect can also be produced by selecting 'None' from the Standard **Shader**'s 'Global Illumination' Inspector property. Self-illuminating materials like these are a useful way to create effects such as neons or other visible light sources.

Emissive materials only directly affect static geometry in your scene. If you need dynamic, or non-static geometry - such as characters, to pick up light from emissive materials, **Light Probes** must be used.

Ambient light

Ambient light is light that is present all around the scene and doesn't come from any specific source object. It can be an important contributor to the overall look and brightness of a scene.

Ambient light can be useful in a number of cases, depending upon your chosen art style. An example would be bright, cartoon-style **rendering** where dark shadows may be undesirable or where lighting is perhaps hand-painted into textures. Ambient light can also be useful if you need to increase the overall brightness of a scene without adjusting individual lights.

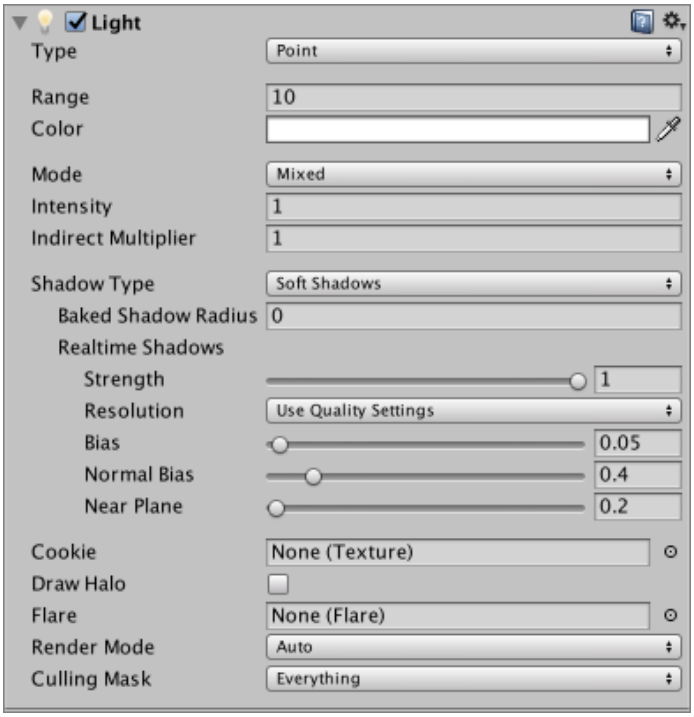
Ambient light settings can be found in the Lighting window.

The Light Inspector

Leave feedback

SWITCH TO SCRIPTING

Lights determine the shading of an object and the shadows it casts. As such, they are a fundamental part of graphical **rendering**. See documentation on lighting and Global Illumination for further details about lighting concepts in Unity.



Properties

Property:	Function:
Type	The current type of light. Possible values are **Directional**, **Point**, **Spot** and **Area** (see the lighting overview for details of these types).
Range	Define how far the light emitted from the center of the object travels (**Point** and **Spot** lights only).
Spot Angle	Define the angle (in degrees) at the base of a spot light's cone (**Spot** light only).
Color	Use the color picker to set the color of the emitted light.
Mode	Specify the Light Mode used to determine if and how a light is "baked". Possible modes are **Realtime**, **Mixed** and **Baked**. See documentation on Realtime Lighting, Mixed Lighting, and Baked Lighting for more detailed information.
Intensity	Set the brightness of the light. The default value for a **Directional** light is 0.5. The default value for a **Point**, **Spot** or **Area** light is 1.
Indirect Multiplier	Use this value to vary the intensity of indirect light. Indirect light is light that has bounced from one object to another. The **Indirect Multiplier** defines the brightness of bounced light calculated by the global illumination (GI) system. If you set **Indirect Multiplier** to a value lower than **1,** the bounced light becomes dimmer with every bounce. A value higher than **1** makes light brighter with each bounce. This is useful, for example, when a dark surface in shadow (such as the interior of a cave) needs to be brighter in order to make detail visible. Alternatively, if you want to use Realtime Global Illumination, but want to limit a single **real-time Light** so that it only emits direct light, set its **Indirect Multiplier** to **0**.
Shadow Type	Determine whether this Light casts Hard Shadows, Soft Shadows, or no shadows at all. See documentation on Shadows for information on hard and soft shadows.
Baked Shadow Angle	If **Type** is set to **Directional** and **Shadow Type** is set to **Soft Shadows**, this property adds some artificial softening to the edges of shadows and gives them a more natural look.

Property:	Function:
Baked Shadow Radius	If **Type** is set to **Point** or **Spot** and **Shadow Type** is set to **Soft Shadows**, this property adds some artificial softening to the edges of shadows and gives them a more natural look.
Realtime Shadows	These properties are available when **Shadow Type** is set to **Hard Shadows** or **Soft Shadows**. Use these properties to control real-time shadow rendering settings.
Strength	Use the slider to control how dark the shadows cast by this Light are, represented by a value between 0 and 1. This is set to 1 by default.
Resolution	Control the rendered resolution of shadow maps. A higher resolution increases the fidelity of shadows, but requires more GPU time and memory usage.
Bias	Use the slider to control the distance at which shadows are pushed away from the light, defined as a value between 0 and 2. This is useful for avoiding false self-shadowing artifacts. See Shadow mapping and the bias property for more information. This is set to 0.05 by default.
Normal Bias	Use the slider to control distance at which the shadow casting surfaces are shrunk along the surface normal, defined as a value between 0 and 3. This is useful for avoiding false self-shadowing artifacts. See documentation on Shadow mapping and the bias property for more information. This is set to 0.4 by default.
Near Plane	Use the slider to control the value for the near clip plane when rendering shadows, defined as a value between 0.1 and 10. This value is clamped to 0.1 units or 1% of the light's **Range** property, whichever is lower. This is set to 0.2 by default.
Cookie	Specify a Texture mask through which shadows are cast (for example, to create silhouettes, or patterned illumination for the Light).
Draw Halo	Tick this box to draw a spherical Halo of light with a diameter equal to the **Range** value. You can also use the Halo **component** to achieve this. Note that the Halo **component** is drawn in addition to the halo from the Light **component**, and that the Halo **component__'s** Size__ parameter determines its radius, not its diameter.
Flare	If you want to set a Flare to be rendered at the Light's position, place an **Asset** in this field to be used as its source.
Render Mode	Use this drop-down to set the rendering priority of the selected Light. This can affect lighting fidelity and performance (see *Performance Considerations,* below).
Auto	The rendering method is determined at run time, depending on the brightness of nearby lights and the current Quality Settings.
Important	The light is always rendered at per-pixel quality. Use **Important** mode only for the most noticeable visual effects (for example, the headlights of a player's car).
Not Important	The light is always rendered in a faster, vertex/object light mode.
Culling Mask	Use this to selectively exclude groups of objects from being affected by the Light. For more information, see Layers.

Details

If you create a Texture that contains an alpha channel and assign it to the **Cookie** variable of the light, the cookie is projected from the light. The cookie's alpha mask modulates the light's brightness, creating light and dark spots on surfaces. This is a great way to add complexity or atmosphere to a **scene**.

All built-in shaders in Unity seamlessly work with any type of light. However, VertexLit **shaders** cannot display Cookies or Shadows.

All Lights can optionally cast Shadows. To do this, set the **Shadow Type** property of each individual Light to either **Hard Shadows** or **Soft Shadows**. See documentation on Shadows for more information.

Directional Light Shadows

See documentation on directional light shadows for an in-depth explanation of how they work. Note that shadows are disabled for directional lights with cookies when **forward rendering** is used. It is possible to write custom shaders to enable shadows in this case; see documentation on writing surface shaders for further details.

Hints

Spot lights with cookies can be extremely effective for creating the effect of light coming in from a window.
Low-intensity point lights are good for providing depth to a Scene.
For maximum performance, use a VertexLit shader. This shader only does per-vertex lighting, giving a much higher throughput on low-end cards.

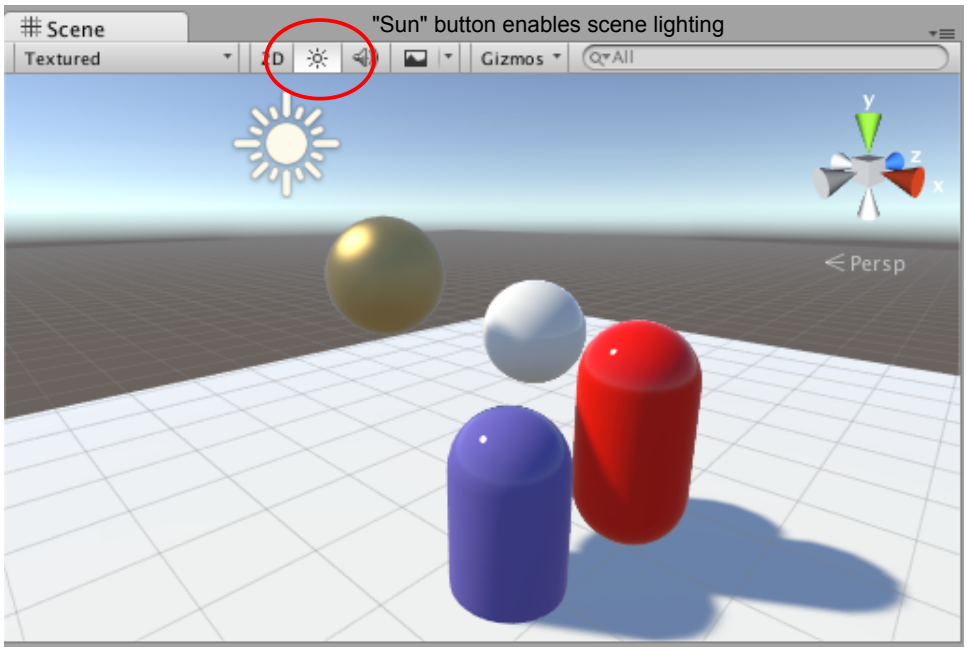
2017–06–08 Page published with limited editorial review

Updated in 5.6

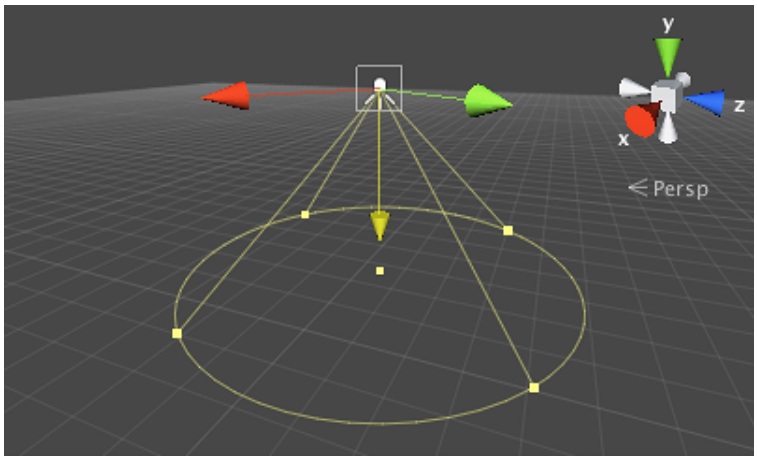
Using Lights

Leave feedback

Lights are very easy to use in Unity - you simply need to create a light of the desired type (eg, from the menu **GameObject > Light > Point Light**) and place it where you want it in the scene. If you enable scene view lighting (the “sun” button on the toolbar) then you can see a preview of how the lighting will look as you move light objects and set their parameters.



A directional light can generally be placed anywhere in the **scene** (except when it is using a Cookie) with the forward/Z axis indicating the direction. A spot light also has a direction but since it has a limited range, its position *does* matter. The shape parameters of spot, point and area lights can be adjusted from the **inspector** or by using the lights’ *Gizmos* directly in the **scene view**.



A spot light with Gizmos visible

Guidelines for Placing Lights

A directional light often represents the sun and has a significant effect on the look of a scene. The direction of the light should point slightly downwards but you will usually want to make sure that it also makes a slight angle with major objects in the scene. For example, a roughly cubic object will be more interestingly shaded and appear to “pop” out in 3D much more if the light isn’t coming head-on to one of the faces.

Spot lights and point lights usually represent artificial light sources and so their positions are usually determined by scene objects. One common pitfall with these lights is that they appear to have no effect at all when you first add them to the scene. This happens when you adjust the range of the light to fit neatly within the scene. The range of a light is the limit at

which the light's brightness dims to zero. If you set, say, a spot light so the base of the cone neatly lands on the floor then the light will have little or no effect unless another object passes underneath it. If you want the level geometry to be illuminated then you should expand point and spot lights so they pass through the walls and floors.

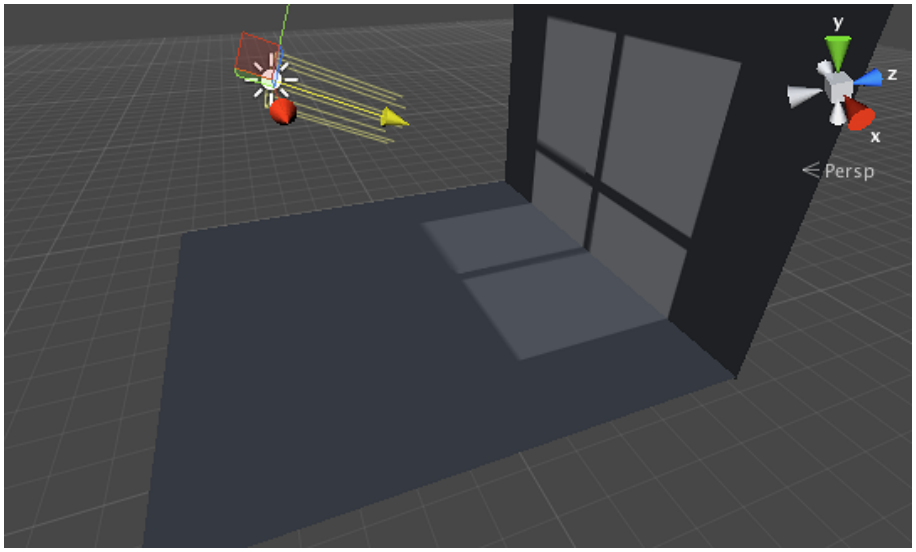
Color and Intensity

A light's color and intensity (brightness) are properties you can set from the inspector. The default intensity and white color are fine for "ordinary" lighting that you use to apply shading to objects but you might want to vary the properties to produce special effects. For example, a glowing green forcefield might be bright enough to bathe surrounding objects in intense green light; car headlights (especially on older cars) typically have a slight yellow color rather than brilliant white. These effects are most often used with point and spot lights but you might change the color of a directional light if, say, your game is set on a distant planet with a red sun.

Cookies

Leave feedback

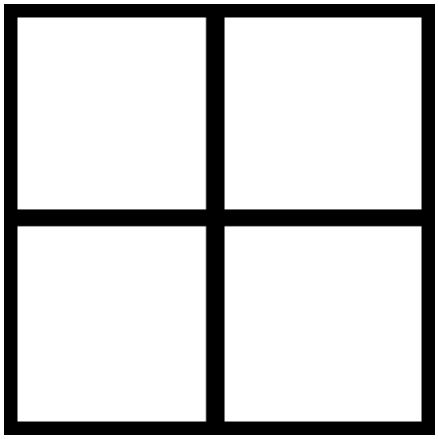
In theatre and film, lighting effects have long been used to create an impression of objects that don't really exist in the set. Jungle explorers may appear to be covered in shadows from an imaginary tree canopy. A prison **scene** often shows the light coming through the barred window, even though the window and indeed the wall are not really part of the set. Though very atmospheric, the shadows are created very simply by placing a shaped mask in between the light source and the action. The mask is known as a *cucoloris* or *cookie* for short. Unity lights allow you to add cookies in the form of textures; these provide an efficient way to add atmosphere to a **scene**.



A directional light cookie simulating light from a window

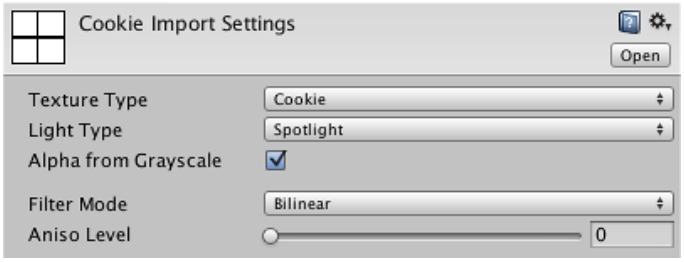
Creating a Cookie

A cookie is just an ordinary texture but only the alpha/transparency channel is relevant. When you import a cookie, Unity gives you the option to convert the brightness of the image to alpha so it is often easier to design your cookie as a grayscale texture. You can use any available image editor to create a cookie and save it to your project's **Assets** folder.

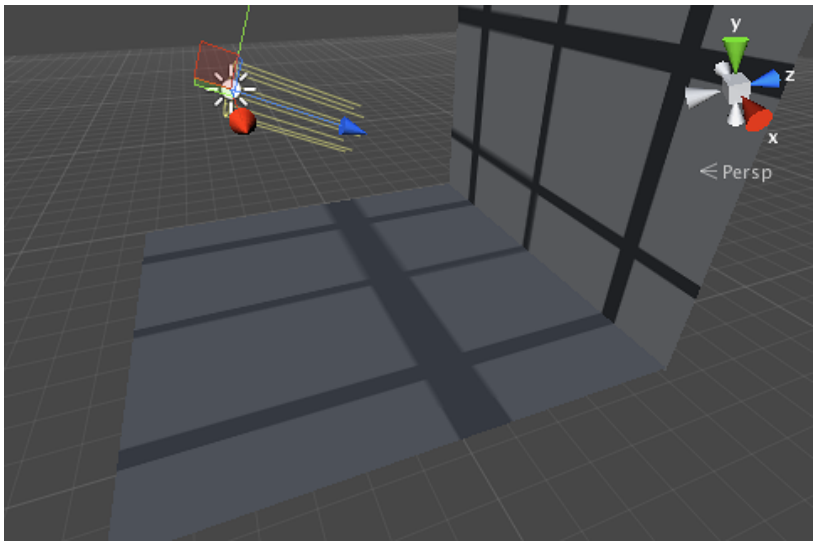


A simple cookie for a window light

When the cookie is imported into Unity, select it from the **Project view** and set the *Texture Type* to *Cookie* in the **inspector**. You should also enable *Alpha From Grayscale* unless you have already designed the image's alpha channel yourself.



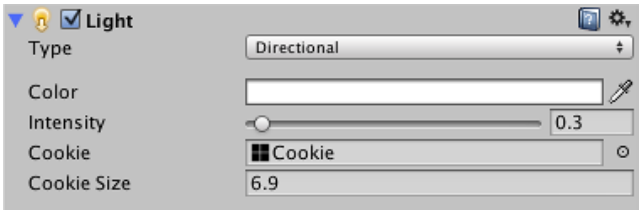
The *Light Type* affects the way the cookie is projected by the light. Since a point light projects in all directions, the cookie texture must be in the form of a **Cubemap**. A spot light should use a cookie with the type set to *Spotlight* but a directional light can actually use either the *Spotlight* or *Directional* options. A directional light with a directional cookie will repeat the cookie in a tiled pattern all over the scene. When a spotlight cookie is used, the cookie will appear just once in the direct path of the "beam" of the light; this is the only case where the position of a directional light is important.



The window cookie "tiled" in directional mode

Applying a Cookie to a Light

When the texture is imported, drag it to the Light's *Cookie* property in the inspector to apply it.



The spot light and point light simply scale the cookie according to the size of the cone or sphere. The directional light has an additional option *Cookie Size* that lets you scale the cookie yourself; the scaling works with both *Spotlight* and *Directional* cookie types.

Uses of Cookies

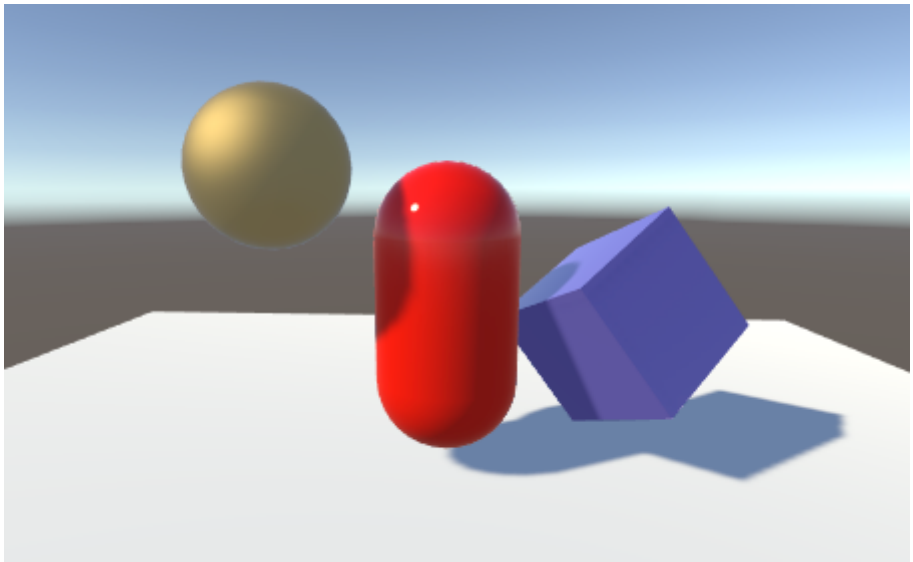
Cookies are often used to change the shape of a light so it matches a detail "painted" in the scene. For example, a dark tunnel may have striplights along the ceiling. If you use standard spot lights for illumination then the beams will have an unexpected round shape but you could use cookies to restrict the lights to a thin rectangle. A monitor screen may cast a green glow onto the face of the character using it but the glow should be restricted to a small box shape.

Note that a cookie need not be completely black and white but can also incorporate any grayscale level. This can be useful for simulating dust or dirt in the path of the light. For example, if a game scene takes place in a long abandoned house, you could add atmosphere by using "dirty" cookies with noise on the windows and other light sources. Similarly, car headlight glass usually contains ridges that create "caustic" patterns of slightly lighter and darker areas in the beam; another good use for a cookie.

Shadows

Leave feedback

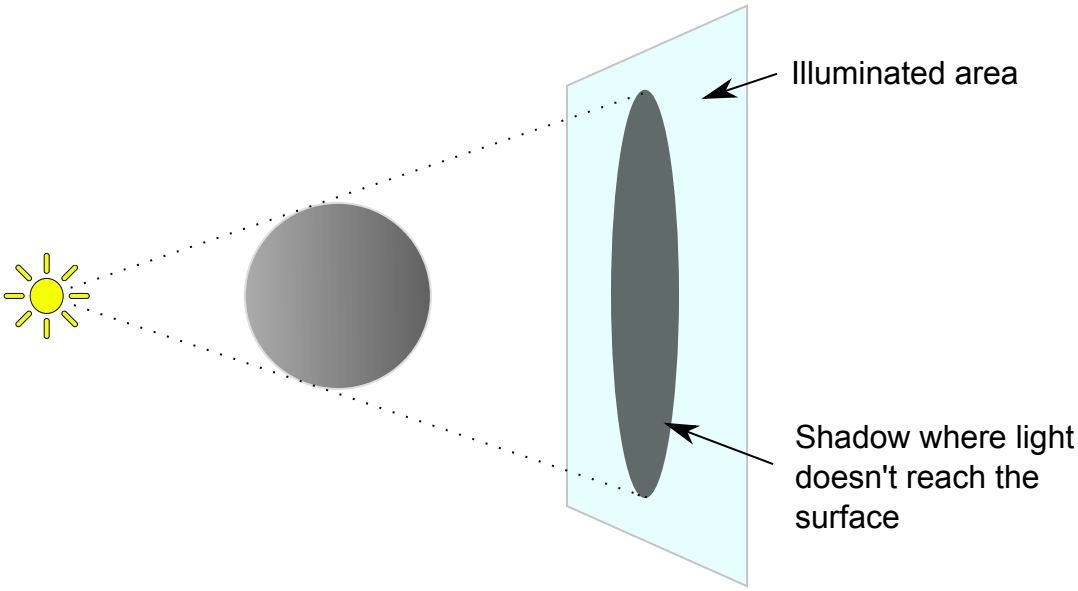
Unity's lights can cast Shadows from an object onto other parts of itself or onto other nearby objects. Shadows add a degree of depth and realism to a scene since they bring out the scale and position of objects that can otherwise look "flat".



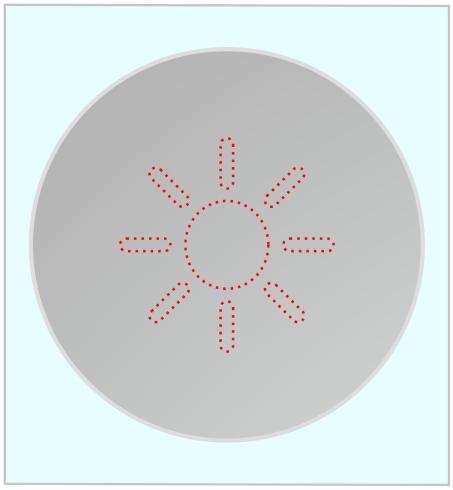
Scene with objects casting shadows

How do shadows work?

Consider the simplest case of a **scene** with a single light source. Light rays travel in straight lines from that source and may eventually hit objects in the **scene**. Once a ray has hit an object, it can't travel any further to illuminate anything else (ie, it "bounces" off the first object and doesn't pass through). The shadows cast by the object are simply the areas that are not illuminated because the light couldn't reach them.



Another way to look at this is to imagine a **camera** at the same position as the light. The areas of the scene that are in shadow are precisely those areas that the **camera** can't see.



A “light’s eye view” of the same scene

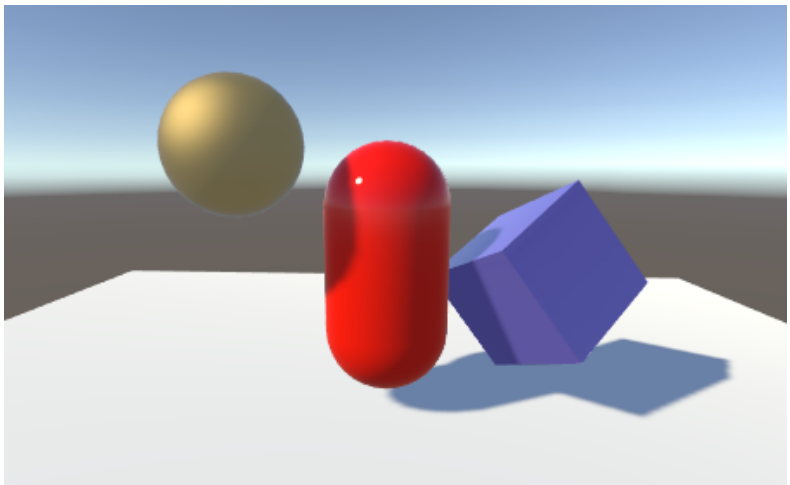
In fact, this is exactly how Unity determines the positions of shadows from a light. The light uses the same principle as a camera to “render” the scene internally from its point of view. A depth buffer system, as used by scene cameras, keeps track of the surfaces that are closest to the light; surfaces in a direct line of sight receive illumination but all the others are in shadow. The depth map in this case is known as a **Shadow Map** (you may find the Wikipedia Page on shadow mapping useful for further information).

The sections below give details on casting shadows from Unity’s Light objects.

Shadows

Leave feedback

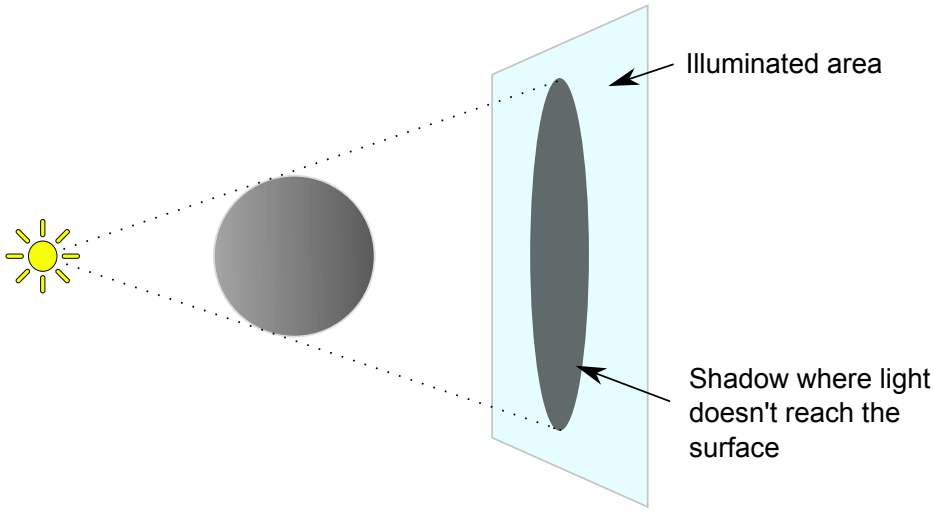
Unity's lights can cast **Shadows** from a GameObject onto other parts of itself or onto other nearby **GameObjects**. Shadows add a degree of depth and realism to a **Scene**, because they bring out the scale and position of **GameObjects** that can otherwise look flat.



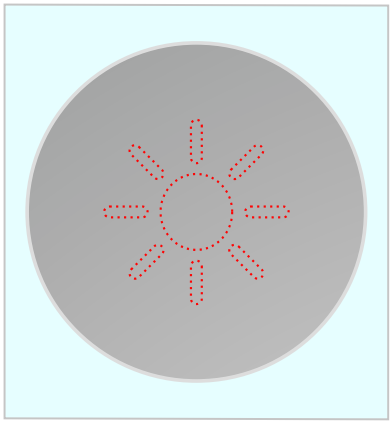
Scene with GameObjects casting shadows

How do Shadows work?

Consider a simple Scene with a single light source. Light rays travel in straight lines from that source, and may eventually hit GameObjects in the Scene. Once a ray has hit a GameObject, it can't travel any further to illuminate anything else (that is, it "bounces" off the first GameObject and doesn't pass through). The shadows cast by the GameObject are simply the areas that are not illuminated because the light couldn't reach them.



Another way to look at this is to imagine a **Camera** at the same position as the light. The areas of the Scene that are in shadow are precisely those areas that the **Camera** can't see.

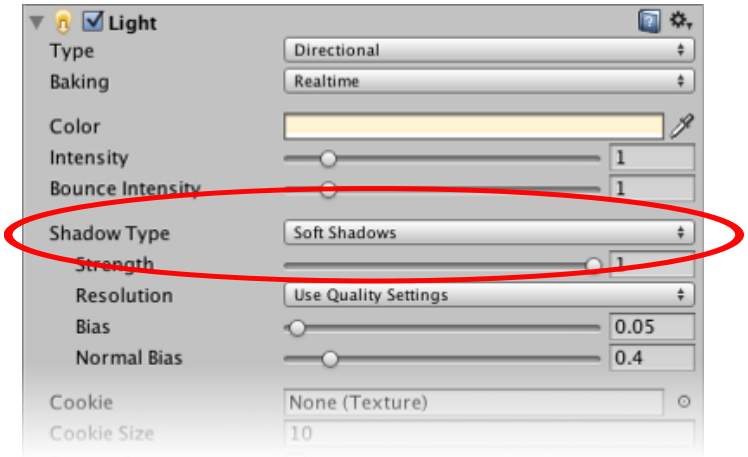


A “light’s eye view” of the same Scene

In fact, this is exactly how Unity determines the positions of shadows from a light. The light uses the same principle as a Camera to “render” the Scene internally from its point of view. A depth buffer system, as used by Scene Cameras, keeps track of the surfaces that are closest to the light; surfaces in a direct line of sight receive illumination but all the others are in shadow. The depth map in this case is known as a **Shadow Map** (you may find the Wikipedia Page on shadow mapping useful for further information).

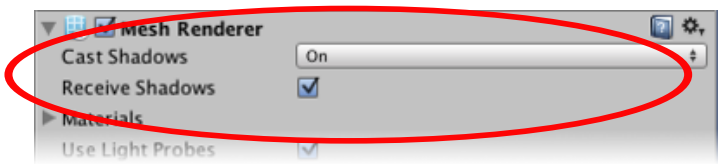
Enabling Shadows

Use the **Shadow Type** property in the **Inspector** to enable and define shadows for an individual light.



Property:	*Function:*
Shadow Type	The **Hard Shadows** setting produces shadows with a sharp edge. Hard shadows are not particularly realistic compared to **Soft Shadows** but they involve less processing, and are acceptable for many purposes. Soft shadows also tend to reduce the “blocky” aliasing effect from the shadow map.
Strength	This determines how dark the shadows are. In general, some light is scattered by the atmosphere and reflected off other GameObjects, so you usually don’t want shadows to be set to maximum strength.
Resolution	This sets the **rendering** resolution for the shadow map’s “Camera” mentioned above. If your shadows have very visible edges, then you might want to increase this value.
Bias	Use this to fine-tune the position and definition of your shadow. See Shadow mapping and the Bias property, below, for more information.
Normal Bias	Use this to fine-tune the position and definition of your shadow. See Shadow mapping and the Bias property, below, for more information.
Shadow Near Plane	This allows you to choose the value for the near plane when rendering shadows. GameObjects closer than this distance to the light do not cast any shadows.

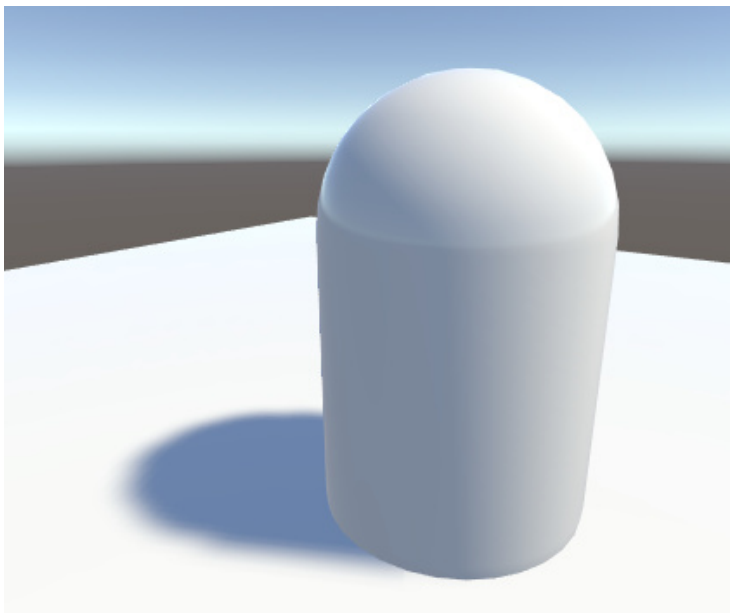
Each Mesh Renderer in the Scene also has a **Cast Shadows** and a **Receive Shadows** property, which must be enabled as appropriate.



Enable **Cast Shadows** by selecting **On** from the drop-down menu to enable or disable shadow casting for the **mesh**. Alternatively, select **Two Sided** to allow shadows to be cast by either side of the surface (so backface culling is ignored for shadow casting purposes), or **Shadows Only** to allow shadows to be cast by an invisible GameObject.

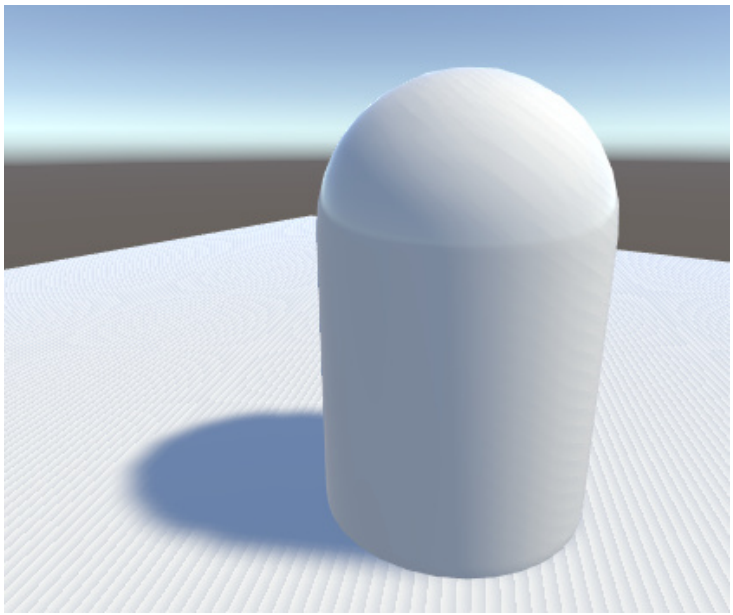
Shadow mapping and the Bias property

The shadows for a given Light are determined during the final Scene rendering. When the Scene is rendered to the main Camera view, each **pixel** position in the view is transformed into the coordinate system of the Light. The distance of a **pixel** from the Light is then compared to the corresponding **pixel** in the shadow map. If the **pixel** is more distant than the shadow map **pixel**, then it is presumably obscured from the Light by another GameObject and it obtains no illumination.



Correct shadowing

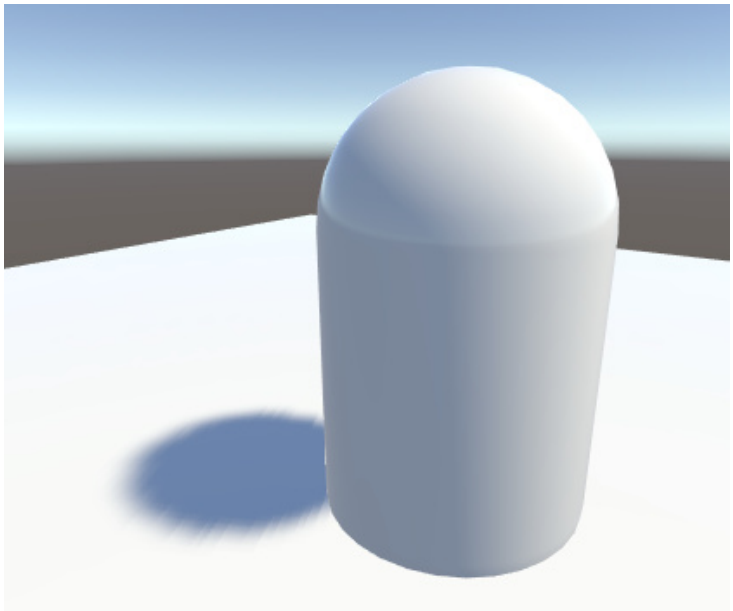
A surface directly illuminated by a Light sometimes appears to be partly in shadow. This is because pixels that should be exactly at the distance specified in the shadow map are sometimes calculated as being further away (this is a consequence of using shadow filtering, or a low-resolution image for the shadow map). The result is arbitrary patterns of pixels in shadow when they should really be lit, giving a visual effect known as “shadow acne”.



Shadow acne in the form of false self-shadowing artifacts

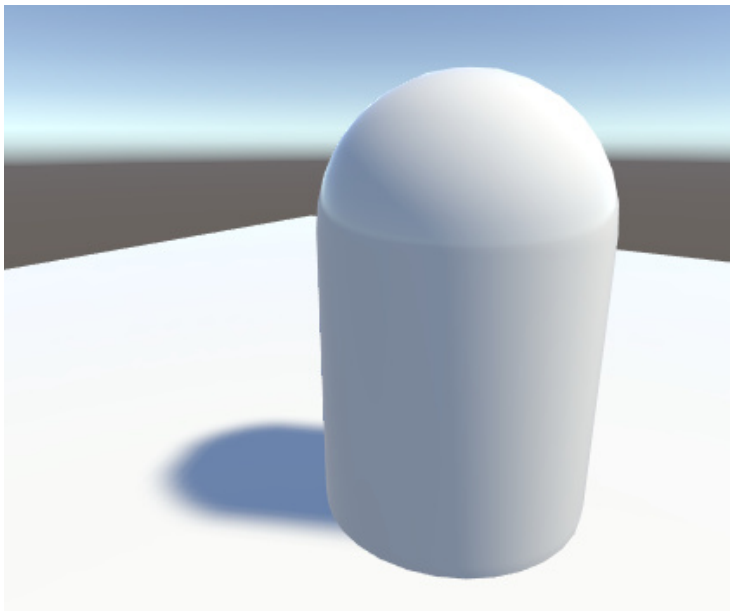
To prevent shadow acne, a **Bias** value can be added to the distance in the shadow map to ensure that pixels on the borderline definitely pass the comparison as they should, or to ensure that while rendering into the shadow map, GameObjects can be inset a little bit along their normals. These values are set by the **Bias** and **Normal Bias** properties in the Light Inspector window when shadows are enabled.

Do not set the **Bias** value too high, because areas around a shadow near the GameObject casting it are sometimes falsely illuminated. This results in a disconnected shadow, making the GameObject look as if it is flying above the ground.



A high **Bias** value makes the shadow appear “disconnected” from the GameObject

Likewise, setting the **Normal Bias** value too high makes the shadow appear too narrow for the GameObject:

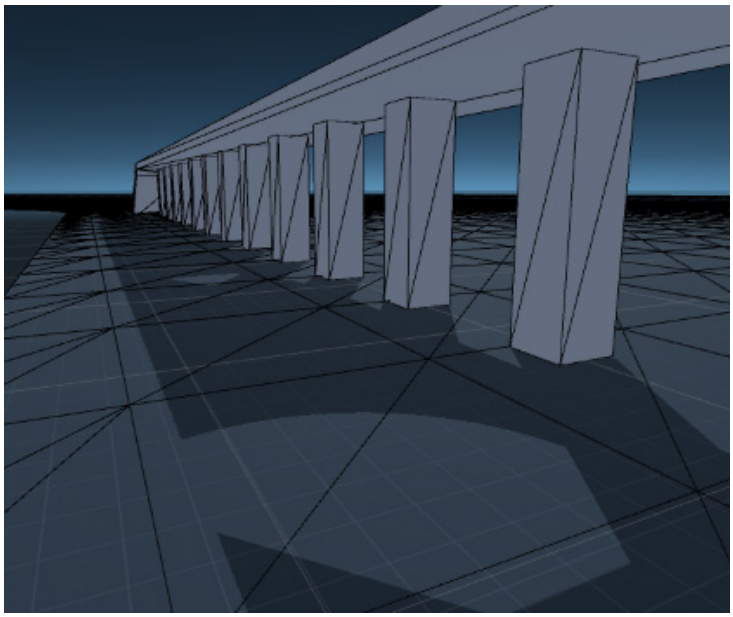


A high **Normal Bias** value makes the shadow shape too narrow

In some situations, **Normal Bias** can cause an unwanted effect called "light bleeding", where light bleeds through from nearby geometry into areas that should be shadowed. A potential solution is to open the GameObject's Mesh Renderer and change the **Cast Shadows** property to **Two Sided**. This can sometimes help, although it can be more resource-instensive and increase performance overhead when rendering the Scene.

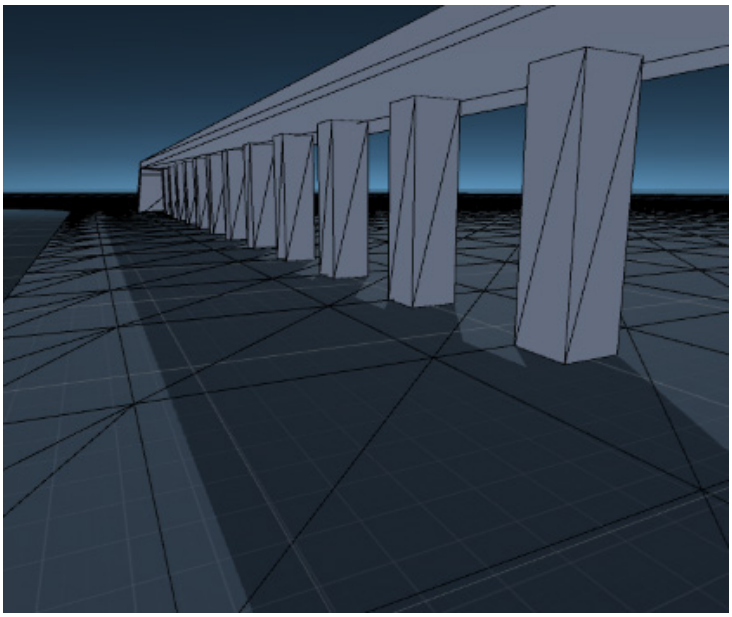
The bias values for a Light may need tweaking to make sure that unwanted effects occur. It is generally easier to gauge the right value by eye rather than attempting to calculate it.

To further prevent shadow acne we are using a technique known as **Shadow pancaking** (see Directional light shadows: Shadow pancaking). This generally works well, but can create visual artifacts for very large triangles.



A low **Shadow near plane offset** value create the appearance of holes in shadows

Tweak the **Shadow Near Plane Offset** property to troubleshoot this problem. Setting this value too high introduces shadow acne.

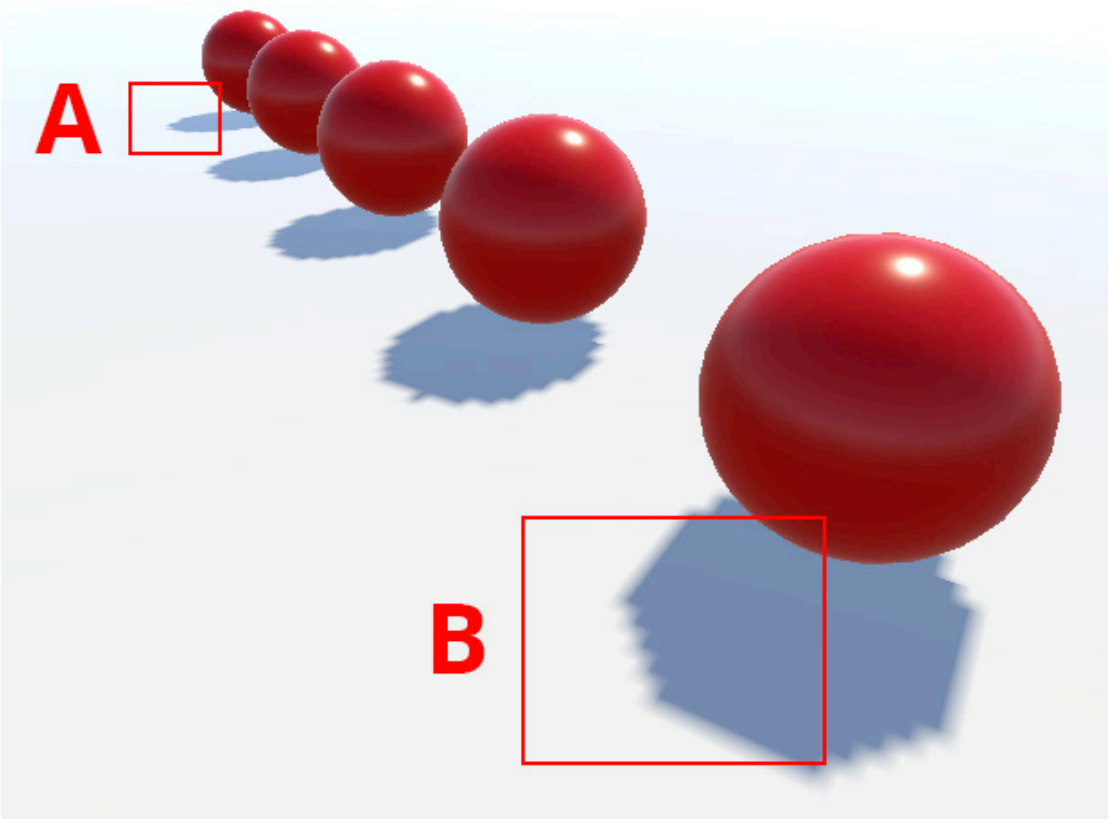


Correct shadowing

Directional light shadows

Leave feedback

A directional light typically simulates sunlight and a single light can illuminate the whole of a **scene**. This means that the shadow map will often cover a large portion of the **scene** at once and this makes the shadows susceptible to a problem called perspective aliasing. Perspective aliasing means that shadow map **pixels** seen close to the **camera** look enlarged and chunky compared to those farther away.

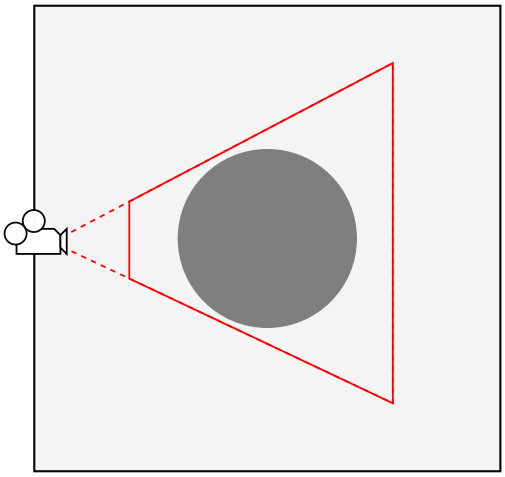


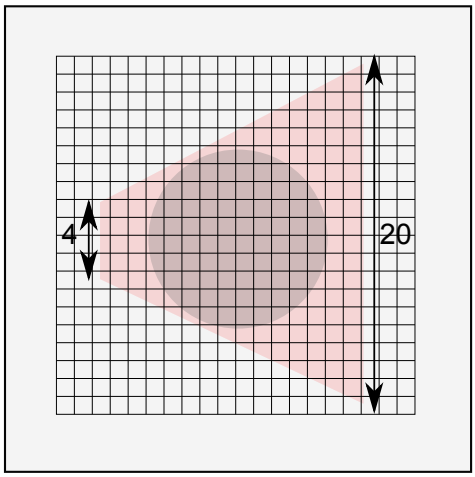
Shadows in the distance (A) have an appropriate resolution, whereas shadows close to camera (B) show perspective aliasing.

Perspective aliasing is less noticeable when you are using soft shadows and a high resolution for the shadow map. However, using these features will increase demands on the graphics hardware and so framerate might suffer.

Shadow cascades

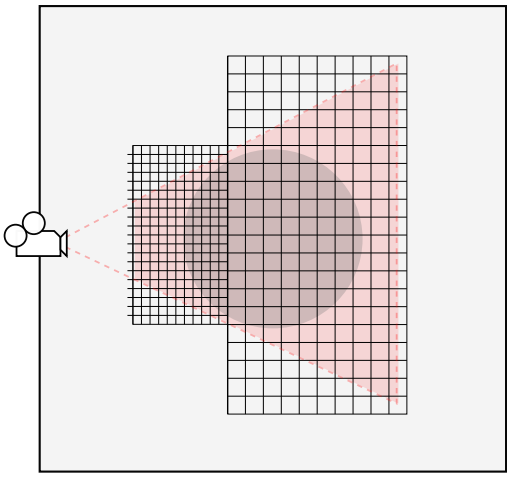
The reason perspective aliasing occurs is that different areas of the shadow map are scaled disproportionately by the camera's perspective. The shadow map from a light needs to cover only the part of the scene visible to the camera, which is defined by the camera's view frustum. If you imagine a simple case where the directional light comes directly from above, you can see the relationship between the frustum and the shadow map.



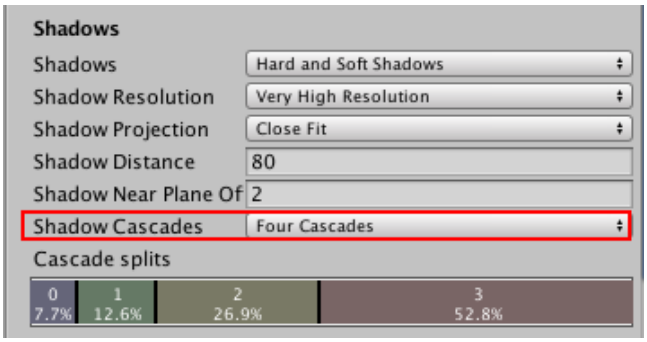


The distant end of the frustum is covered by 20 pixels of shadow map while the near end is covered by only 4 pixels. However, both ends appear the *same size* onscreen. The result is that the resolution of the map is effectively much less for shadow areas that are close to the camera. (Note that in reality, the resolution is much higher than 20x20 and the map is usually not perfectly square-on to the camera.)

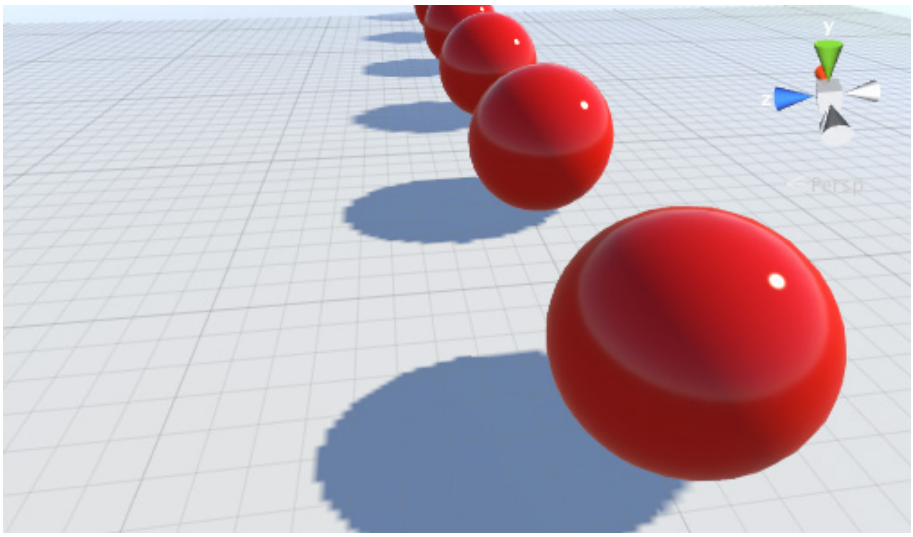
Using a higher resolution for the whole map can reduce the effect of the chunky areas but this uses up more memory and bandwidth while **rendering**. You will notice from the diagram, though, that a large part of the shadow map is wasted at the near end of the frustum because it will never be seen; also shadow resolution far away from the camera is likely to be too high. It is possible to split the frustum area into two zones based on distance from the camera. The zone at the near end can use a separate shadow map at a reduced size (but with the same resolution) so that the number of pixels is evened out somewhat.



These staged reductions in shadow map size are known as **cascaded shadow maps** (sometimes called "Parallel Split Shadow Maps"). From the Quality Settings, you can set zero, two or four cascades for a given quality level.



The more cascades you use, the less your shadows will be affected by perspective aliasing, but increasing the number does come with a rendering overhead. However, this overhead is still less than it would be if you were to use a high resolution map across the whole shadow.



Shadow from the earlier example with four cascades

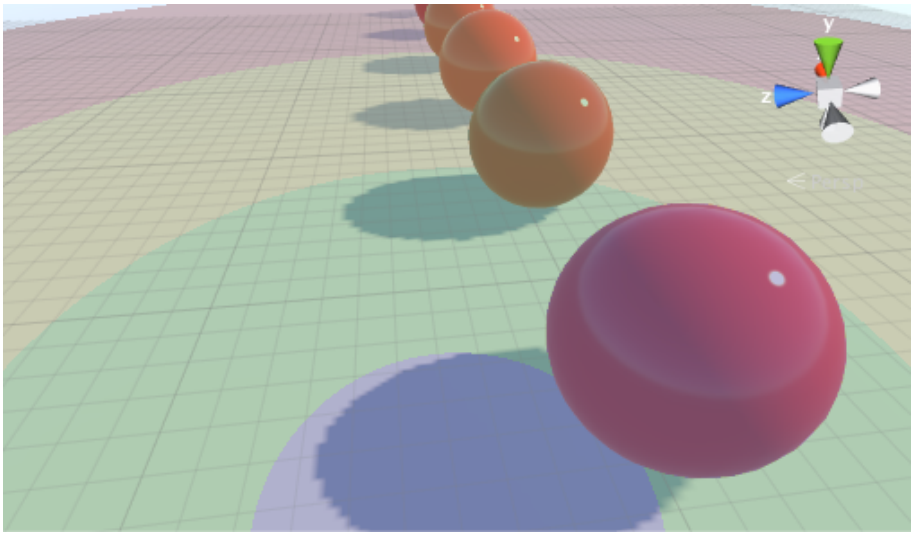
Shadow distance

Shadows from objects tend to become less noticeable the farther the objects are from the camera; they appear smaller onscreen and also, distant objects are usually not the focus of attention. Unity lets you take advantage of this effect by providing a *Shadow Distance* property in the Quality Settings. Objects beyond this distance (from the camera) cast no shadows at all, while the shadows from objects approaching this distance gradually fade out.

Set the shadow distance as low as possible to help improve rendering performance. This works because distant objects do not need to be rendered into the shadow map at all. Additionally, the Scene often looks better with distant shadows removed. Getting the shadow distance right is especially important for performance on mobile platforms, because they don't support shadow cascades. If the current camera far plane is smaller than the shadow distance, Unity uses the camera far plane instead of the shadow distance.

Visualising shadow parameter adjustments

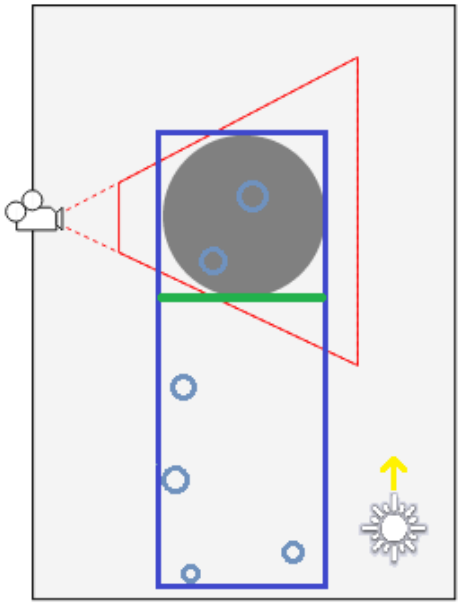
The **Scene View** has a draw mode called **Shadow Cascades**, which uses coloration to show the parts of the Scene using different cascade levels. Use this to help you get the shadow distance, cascade count and cascade split ratios just right. Note that this visualization use the Scene view far plane, which is usually bigger than the shadow distance, so you might need to lower the Shadow distance if you want to match the in-game behavior of the Camera with a small far plane.



Shadow Cascades draw mode in the Scene View

Shadow Pancaking

To further prevent shadow acne we are using a technique known as **Shadow pancaking**. The idea is to reduce the range of the light space used when rendering the shadow map along the light direction. This lead to an increased precision in the shadow map, reducing shadow acne.

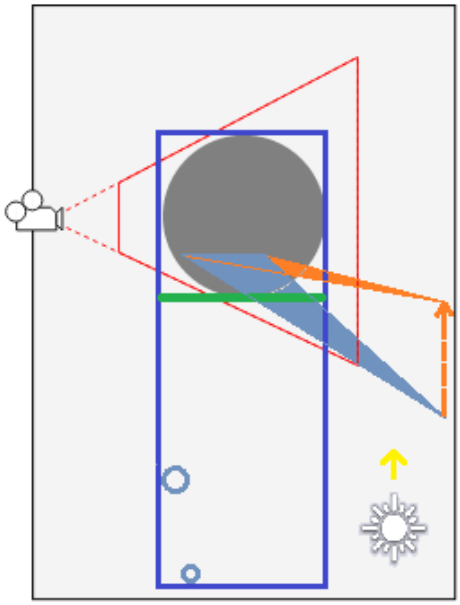


A diagram showing the shadow pancaking principle

In the above digram:

The **light blue circles** represent the shadow casters
The **dark blue rectangle** represents the original light space
The **green line** represents the optimized near plane (excluding any shadow casters not visible in the view frustum)

Clamp these shadow casters to the near clip plane of the optimized space (in the Vertex Shader). Note that while this works well in general, it can create artifacts for very large triangles crossing the near clip plane:



Large triangle problem

In this case, only one vertex of the blue triangle is behind the near clip plane and gets clamped to it. However, this alters the triangle shape, and can create incorrect shadowing.

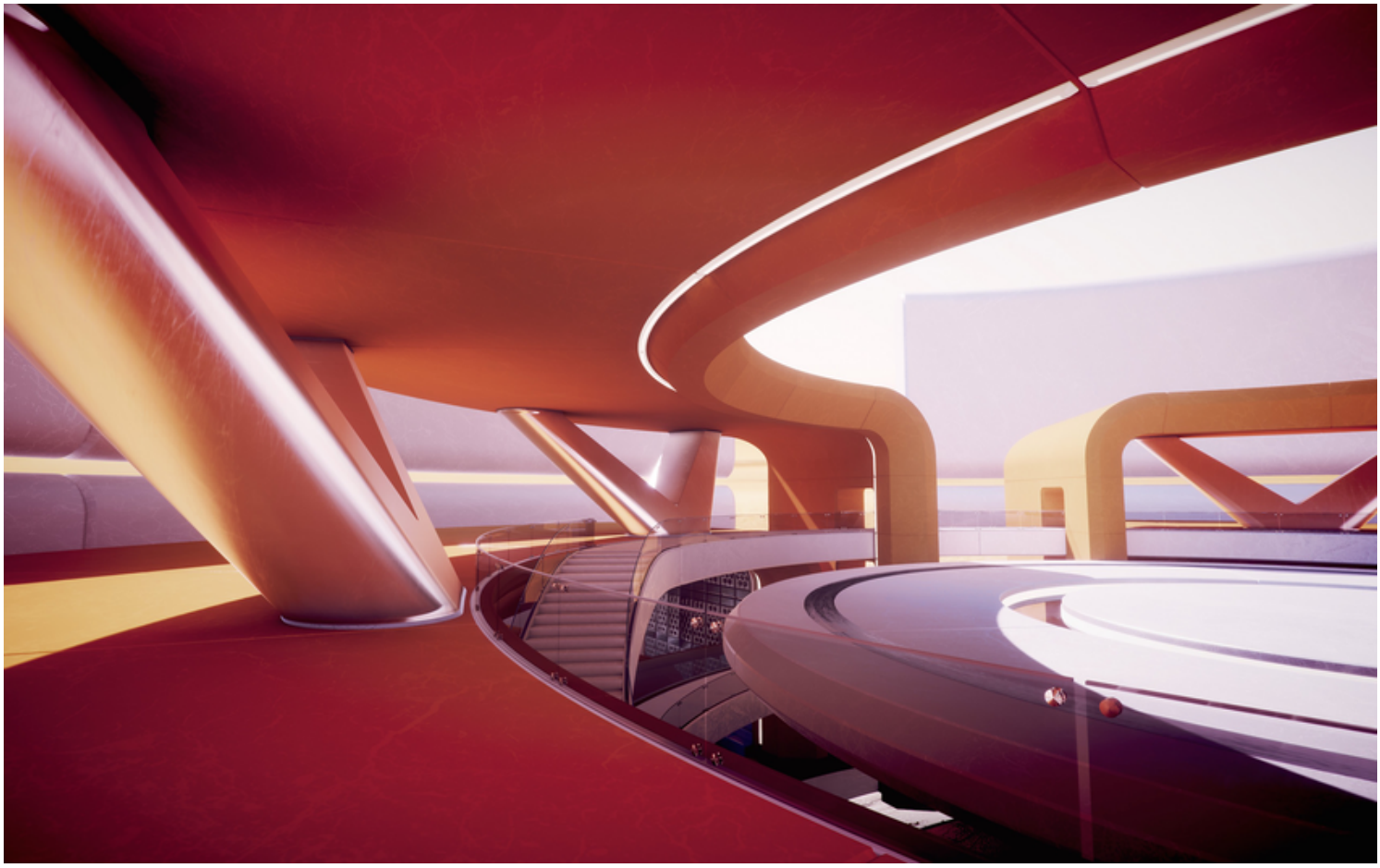
You can tweak the **Shadow Near Plane Offset** property from the Quality Settings to avoid this problem. This pulls back the near clip plane. However, setting this value very high eventually introduces shadow acne, because it raises the range that the

shadow map needs to cover in the light direction. Alternatively, you can also tesselate the problematic shadow casting triangles. See the **bias** section in Shadow Overview for more information.

Global Illumination

Leave feedback

Global Illumination (GI) is a system that models how light is bounced off of surfaces onto other surfaces (indirect light) rather than being limited to just the light that hits a surface directly from a light source (direct light). Modelling indirect lighting allows for effects that make the virtual world seem more realistic and connected, since objects affect each other's appearance. One classic example is 'color bleeding' where, for example, sunlight hitting a red sofa will cause red light to be bounced onto the wall behind it. Another is when sunlight hits the floor at the opening of a cave and bounces around inside so the inner parts of the cave are illuminated too.



Global illumination in the **Scene View**. Note the subtle effect of indirect lighting.

GI concepts

Traditionally, video games and other realtime graphics applications have been limited to direct lighting, while the calculations required for indirect lighting were too slow so they could only be used in non-realtime situations such as CG animated films. A way for games to work around this limitation is to calculate indirect light only for objects and surfaces that are known ahead of time to not move around (that are static). That way the slow computation can be done ahead of time, but since the objects don't move, the indirect light that is pre-calculated this way will still be correct at runtime. Unity supports this technique, called Baked GI (also known as Baked Lightmaps), which is named after "the bake" - the process in which the indirect light is precalculated and stored (baked). In addition to indirect light, Baked GI also takes advantage of the greater computation time available to generate more realistic soft shadows from area lights and indirect light than what can normally be achieved with realtime techniques.

Additionally, Unity 5.0 adds support for a new technique called Precomputed Realtime GI. It still requires a precomputation phase similar to the bake mentioned above, and it is still limited to static objects. However it doesn't just precompute how light bounces in the **scene** at the time it is built, but rather it precomputes all possible light bounces and encodes this information for use at runtime. So essentially for all static objects it answers the question "if any light hits this surface, where does it bounce to?" Unity then saves this information about which paths light can propagate by for later use. The final lighting is done at runtime by feeding the actual lights present into these previously computed light propagation paths.

This means that the number and type of lights, their position, direction and other properties can all be changed and the indirect lighting will update accordingly. Similarly it's also possible to change material properties of objects, such as their color, how much light they absorb or how much light they emit themselves.

While Precomputed Realtime GI also results in soft shadows, they will typically have to be more coarse-grained than what can be achieved with Baked GI unless the scene is very small. Also note that while Precomputed Realtime GI does the final lighting at runtime, it does so iteratively over several frames, so if a big a change is done in the lighting, it will take more frames for it to fully take effect. And while this is fast enough for realtime applications, if the target platform has very constrained resources it may be better to to use Baked GI for better runtime performance.

Limitations of GI

Both Baked GI and Precomputed Realtime GI have the limitation that only static objects can be included in the bake/precomputation - so moving objects cannot bounce light onto other objects and vice versa. However they can still pick up bounce light from static objects using Light Probes. Light Probes are positions in the scene where the light is measured (probed) during the bake/precomputation, and then at runtime the indirect light that hits non-static objects is approximated using the values from the probes that the object is closest to at any given moment. So for example a red ball that rolls up next to a white wall would not bleed its color onto the wall, but a white ball next to a red wall could pick up a red color bleed from the wall via the light probes.

Examples of GI effects

Changing the direction and color of a directional light to simulate the effect of the sun moving across the sky. By modifying the **skybox** along with the directional light it is possible to create a realistic time-of-day effect that is updated at runtime. (In fact the new built-in procedural skybox makes it easy to do this).

As the day progresses the sunlight streaming in through a window moves across the floor, and this light is realistically bounced around the room and onto the ceiling. When the sunlight reaches a red sofa, the red light is bounced onto the wall behind it. Changing the color of the sofa from red to green will result in the color bleed on the wall behind it turning from red to green too.

Animating the emissiveness of a neon sign's material so it starts glowing onto its surroundings when it is turned on.

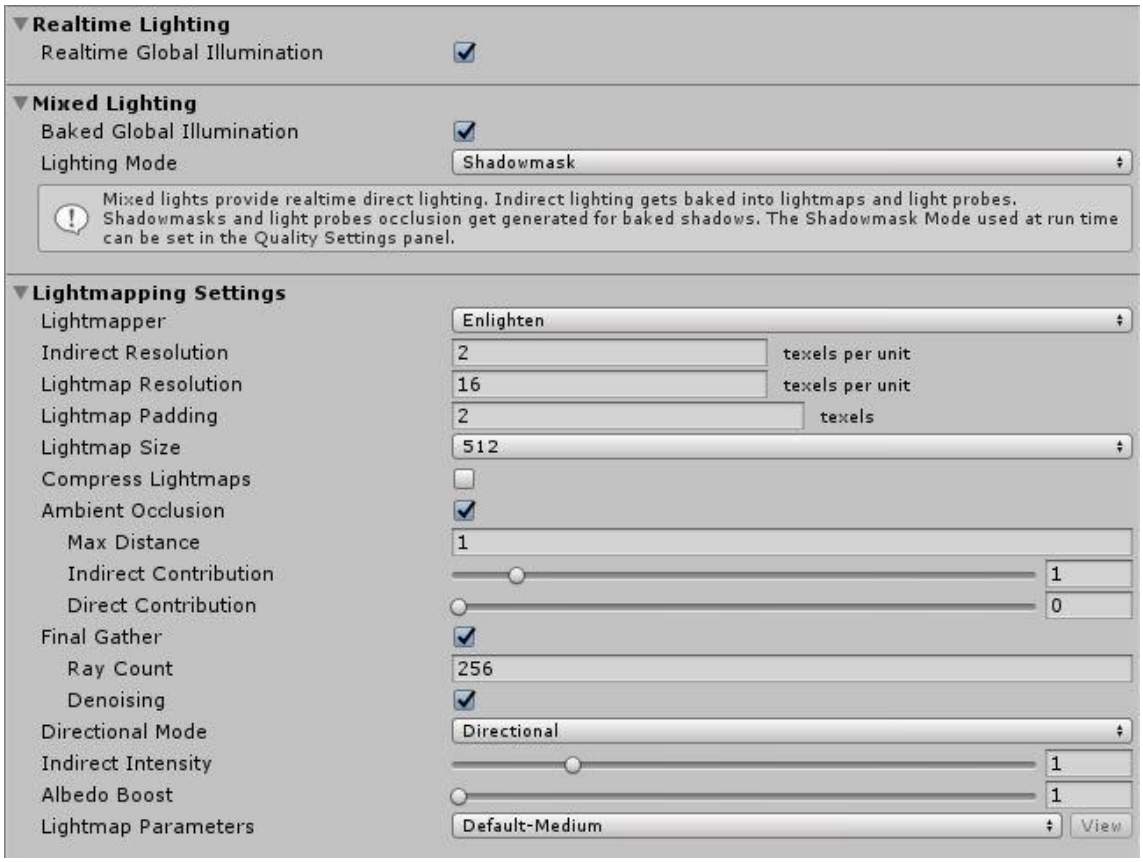
The following sections go into detail about how to use this feature.

Enlighten

Leave feedback

Unity provides two distinct techniques for precomputing global illumination (GI) and bounced lighting. These are **Baked Global Illumination** and **Precomputed Realtime Global Illumination**. The Enlighten lighting system provides solutions for both.

To find the following settings, navigate to Unity's top menu and go to **Window** > **Rendering** > **Lighting**.



Property:	Function:
Realtime Global Illumination	Makes Unity calculate and update lighting in real time. For more information, see documentation on Realtime Global Illumination, and the Unity tutorial on Precomputed Realtime GI.
Lighting Mode	Specifies which lighting mode to use for all mixed lights in the **Scene**. Options are Baked Indirect, Distance Shadowmask, Shadowmask, and Subtractive.
Lightmapper	Specifies which internal lighting calculation software to use to calculate **lightmaps** in the Scene. The options are **Progressive** and **Enlighten**. The default value is **Progressive**; set it to **Enlighten** to use the system described in this page. If you want to use the **Progressive** system, see documentation on the Progressive Lightmapper.
Indirect Resolution	This property is only available when **Realtime Global Illumination** is enabled. Use this value to specify the number of texels per unit to use for indirect lighting calculations. Increasing this value improves the visual quality of indirect light, but also increases the time it takes to bake lightmaps. The default value is 2. See the Unity tutorial on Realtime Resolution for details about Indirect Resolution.
Lightmap Resolution	Specifies the number of texels per unit to use for lightmaps. Increasing this value improves lightmap quality, but also increases bake times. The default value in a new Scene is 40.
Lightmap Padding	Specifies the separation (in texel units) between separate shapes in the baked lightmap. The default value is 2.
Lightmap Size	The size (in pixels) of the full lightmap texture, which incorporates separate regions for the individual object textures. The default value is 1024.

Property:	Function:
Compress Lightmaps	Compresses lightmaps so that they require less storage space. However, the **compression** process can introduce unwanted visual effects into the texture. This property is checked by default.
Ambient Occlusion	Opens a group of settings which allow you to control the relative brightness of surfaces in ambient occlusion. Higher values indicate a greater contrast between the occluded and fully lit areas. This only applies to the indirect lighting calculated by the GI system. This property is enabled by default.
Max Distance	Sets a value to control how far the lighting system casts rays in order to determine whether or not to apply occlusion to an object. A larger value produces longer rays and contributes more shadows to the lightmap, while a smaller value produces shorter rays that contribute shadows only when objects are very close to one another. A value of 0 casts an infinitely long ray that has no maximum distance. The default value is 1.
Indirect Contribution	Use the slider to scale the brightness of indirect light (that is, ambient light, or light bounced and emitted from objects) in the final lightmap, from a value between 0 and 10. The default value is 1. Values less than 1 reduce the intensity, and values greater than 1 increase it.
Direct Contribution	Use the slider to scale the brightness of direct light from a value between 0 and 10. The default value is 0. The higher this value is, the greater the contrast applied to the direct lighting.
Final Gather	Enable this if you want Unity to calculate the final light bounce in the GI calculation at the same resolution as the baked lightmap. This improves the visual quality of the lightmap, but at the cost of additional baking time in the Editor.
Ray Count	Defines the number of rays emitted for each final gather point. The default value is 256.
Denoising	Applies a de-noising filter to the **Final Gather** output. This property is checked by default.
Directional Mode	You can set up the lightmap to store information about the dominant incoming light at each point on the surfaces of your **GameObjects**. See documentation on Directional Lightmapping for further details. The default mode is **Directional**.
Directional	In **Directional** mode, Unity generates a second lightmap to store the dominant direction of incoming light. This allows diffuse normal-mapped materials to work with the GI. Directional mode requires about twice as much storage space for the additional lightmap data. Unity cannot decode directional lightmaps on SM2.0 hardware or when using GLES2.0. They fall back to non-directional lightmaps.
Non-directional	In **Non-directional** mode, Unity does not generate a second lightmap for the dominant direction of incoming light, and instead stores all lighting information in the same place.
Indirect Intensity	Controls the brightness of the indirect light that Unity stores in real-time and baked lightmaps, from a value between 0 and 5. A value above 1 increases the intensity of indirect light, and a value of less than 1 reduces indirect light intensity. The default value is 1.
Albedo Boost	Controls the amount of light Unity bounces between surfaces by intensifying the albedo of Materials in the Scene, from a value between 1 and 10. Increasing this draws the albedo value towards white for indirect light computation. The default value is 1, which is physically accurate.
Lightmap Parameters	Unity uses a set of general parameters for the lightmapping in addition to Lighting window properties of the. A few defaults are available from the menu for this property, but you can also use the **Create New** option to create your own lightmap parameter file. See the Lightmap Parameters page for further details. The default value is **Default-Medium**.

See the Precomputed Realtime GI tutorial to learn more about Enlighten optimizations.

Progressive Lightmapper added in 2018.1

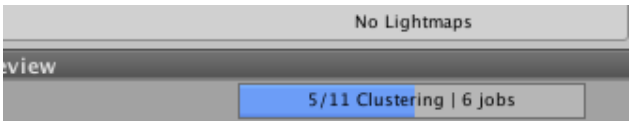
2017–05–18 Page published with limited editorial review

Using precomputed lighting

Leave feedback

In Unity, precomputed lighting is calculated in the background either as an automatic process or when manually initiated. In either case, it is possible to continue working in the editor while these processes run behind the scenes.

When the precompute process is running, a blue progress bar will appear in the bottom right of the Editor. There are different stages which need to be completed depending on whether Baked GI or Precomputed Realtime GI is enabled. Information on the current process is shown on-top of the progress bar.



Progress bar showing the current state of Unity's precompute.

In the example above, we can see that we are at task 5 of 11 which is, 'Clustering' and there are 6 jobs remaining before that task is complete and the precompute moves on to task 6. The various stages are listed below:

Precomputed Realtime GI

Create Geometry
Layout Systems
Create Systems
Create Atlas
Clustering
Visibility
Light Transport
Tetrahedralize Probes
Create ProbeSet
Probes

Ambient Probes
Baked/Realtime Ref. Probes
Baked GI

Create Geometry
Atlassing
Create Baked Systems
Baked Resources
Bake AO
Export Baked Texture
Bake Visibility
Bake Direct
Ambient and Emissive
Create Bake Systems
Bake Runtime
Upsampling Visibility
Bake Indirect
Final Gather
Bake ProbesSet
Compositing

Starting a precompute

Only static geometry is considered by Unity's precomputed lighting solutions. To begin the lighting precompute process we need at least one GameObject marked as 'static' in our scene. This can either be done individually, or by shift-selecting multiple **GameObjects** from the hierarchy panel.

From the Inspector panel, the Static checkbox can be selected (**Inspector > Static**). This will set all of the GameObject's 'static options', or 'flags', including navigation and batching, to be static, which may not be desirable. For Precomputed Realtime GI, only 'Lightmap Static' needs to be checked.

For more fine-grained control, individual static options can be set from the drop-down list accessible to the right of the Static checkbox in the **Inspector** panel. Additionally, objects can also be set to Static in the Object area of the lighting window.

If you set your Scene to **Auto Generate** (menu: **Window** > **Rendering** > **Lighting Settings** > **Scene** > **Auto Generate**), Unity's lighting precompute begins automatically, and updates automatically when any static geometry in your Scene changes. If you do not enable **Auto Generate**, you must start the lighting precompute manually.

Auto/manual precompute

If you have checked **Auto Generate** in the bottom of Unity's Lighting panel (menu: **Window** > **Rendering** > **Lighting Settings** > **Scene** > **Auto Generate**), this precompute begins automatically as a background process whenever the static geometry within your Scene changes.

However, if you have not enabled **Auto Generate**, you must manually start the lighting precompute process by clicking the 'Build' button next to it. This begins the precompute in much the same way, and gives you control over when this process starts.

Auto Generate can be useful when working on smaller or less complex Scenes, because it quickly produces accurate lighting results while you move or editing static GameObjects in your Scene. However, when working on large or complex Scenes, you might prefer to switch to the manual option, so that your computer is not running a high CPU usage and repeatedly re-starting the lighting precompute each time you modify your Scene.

When you manually initiate a precompute, all aspects of your Scene lighting are evaluated and computed. To recalculate and bake just the Reflection Probes by themselves, click the drop-down menu attached to the **Generate Lighting** button (menu: **Window** > **Lighting Settings** > **Scene** > **Generate Lighting**) and select **Bake Reflection Probes**.

NOTE: When using **Auto Generate** mode, Unity stores your lighting data in a temporary cache with a limited size. This means that when you exceed the cache's size, Unity deletes old lighting data. A problem might occur when building your project if some of your Scenes rely on auto-generated lighting data that has been deleted. In this case, your Scenes might not have the correct lighting in the built project. Therefore, before building your game, you should uncheck **Auto Generate,** and generate the lighting data manually for all your Scenes. Unity then saves your lighting data as **Asset** files in your project folder, which means you have the data saved as part of your project and included in your build.

Enabling Baked GI or Realtime GI

Unity enables both **Baked GI** and **Realtime GI** by default. **Baked GI** is all precomputed; **Realtime GI** carries out some precomputation when indirect lighting is used. With both enabled, you then use each individual Light in your Scene to control which GI system it should use (in the Light component, use the **Mode** setting to do this). See documentation on the Lighting window and Global Illumination to learn more.

The most flexible way to use the lighting system is to use **Baked GI** and **Realtime GI** together. However, this is also the most performance-heavy option. To make your game less resource-intensive, you can choose to disable **Realtime GI** or **Baked GI**. Note that doing this this reduces the flexibility and functionality of your lighting system.

To manually enable or disable Global Illumination, open the Lighting window (**Window** > **Lighting Settings** > **Scene**). Tick **Realtime Global Illumination** to enable **Realtime GI**, and tick **Baked Global Illumination** to enable **Baked GI**. Untick these checkboxes to disable the respective GI system. If any Lights are set to the mode you have disabled, they are are overridden and set to the active GI system.

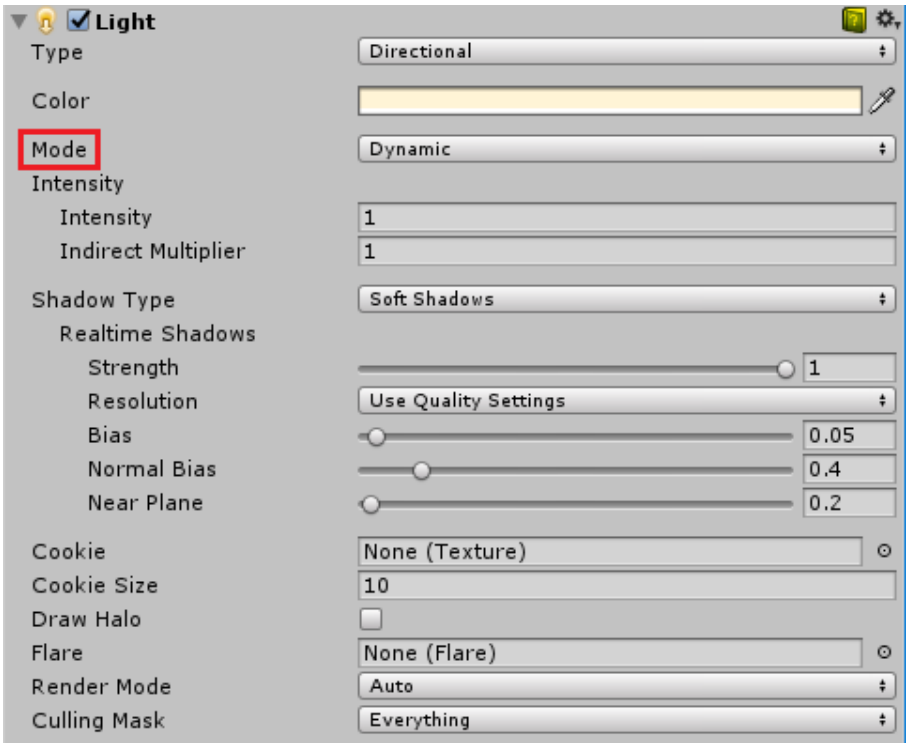
Per-light settings

To set properties for each individual Light, select it in the Scene or Hierarchy window, then edit the settings on the Light component in the Inspector window.

The default **Mode** for each light is **Dynamic**. This means that the Light contributes direct lighting to your Scene, and Unity's **Realtime GI** handles indirect lighting.

If you set the Light's **Mode** to **Static,** then that Light only contributes lighting to Unity's **Baked GI** system. Both direct and indirect lighting from those Lights are baked into light maps, and cannot be changed during gameplay.

If you set the Light's **Mode** to **Stationary**, GameObjects marked as **Static** still include this light in their **Baked GI** light maps. However, unlike Lights marked as **Static**, **Stationary** Lights still contribute real-time lighting, based on the stationary bake mode in the Lighting window (menu: **Window** > **Lighting Settings**). This is useful if you are using light maps in a static environment, but you still want a good integration between dynamic and light map static geometry.



A Directional Light with the **Mode** set to **Dynamic**.
See documentation on Lighting Modes for more details.

GI cache

In either Baked GI or Precomputed Realtime GI, Unity caches (stores) data about your scene lighting in the 'GI Cache', and will try to reuse this data whenever possible to save time during precompute. The number and nature of the changes you have made to your scene will determine how much of this data can be reused, if at all.

This cache is stored outside of your Unity project and can be cleared using (**Preference** > **GI Cache** > **Clear Cache**). Clearing this means that all stages of the precompute will need to be recalculated from the beginning and this can

therefore be time consuming. However in some cases, where perhaps you need to reduce disk usage, this may be helpful.

LOD for baked GI

Level-of-detail is taken into consideration when Unity generates baked **lightmaps**. Direct lighting is computed using the actual surfaces of all **LODs**. Lower LOD levels use **light probes** to fetch indirect lighting. The resulting lighting is baked into lightmap.

This means that you should place light probes around your LODs to capture indirect lighting. The object will not use lightprobes at runtime if you use fully baked GI.

2017–06–08 Page published with limited editorial review

Light Modes added in 5.6

LOD and Realtime GI

Leave feedback

Read this page before you use Realtime Global Illumination (GI) on models which use the LOD (level of detail) feature.

When you use Unity's **LOD** system in a **Scene** with baked lighting and Realtime GI, the system lights the most detailed model out of the **LOD Group** as if it is a regular static model. It uses **lightmaps** for direct and indirect lighting, and separate **lightmaps** for Realtime GI.

To allow the baking system to produce real-time or baked lightmaps, select the **GameObject** you want to affect, view its Renderer **component** in the **Inspector** window, and check that **Lightmap Static** is enabled.

For lower LODs in an LOD Group, you can only combine baked lightmaps with Realtime GI from Light Probes or Light Probe Proxy Volumes, which you must place around the LOD Group.

Whenever you use Realtime GI, the Renderer enables the **Light Probes** option for the lower LODs, even if you enable **Lightmap Static**.

An animation showing how real-time ambient color affects the Realtime GI used by lower LODs

2017–10–25 Page amended with editorial review

Added Realtime Global Illumination in 2017.3

Progressive Lightmapper

Leave feedback

Progressive Lightmapper is a fast path-tracing-based lightmapper system that provides baked **lightmaps** and **Light Probes** with progressive updates in the Editor. It requires non-overlapping UVs with small area and angle errors, and sufficient padding between the charts.

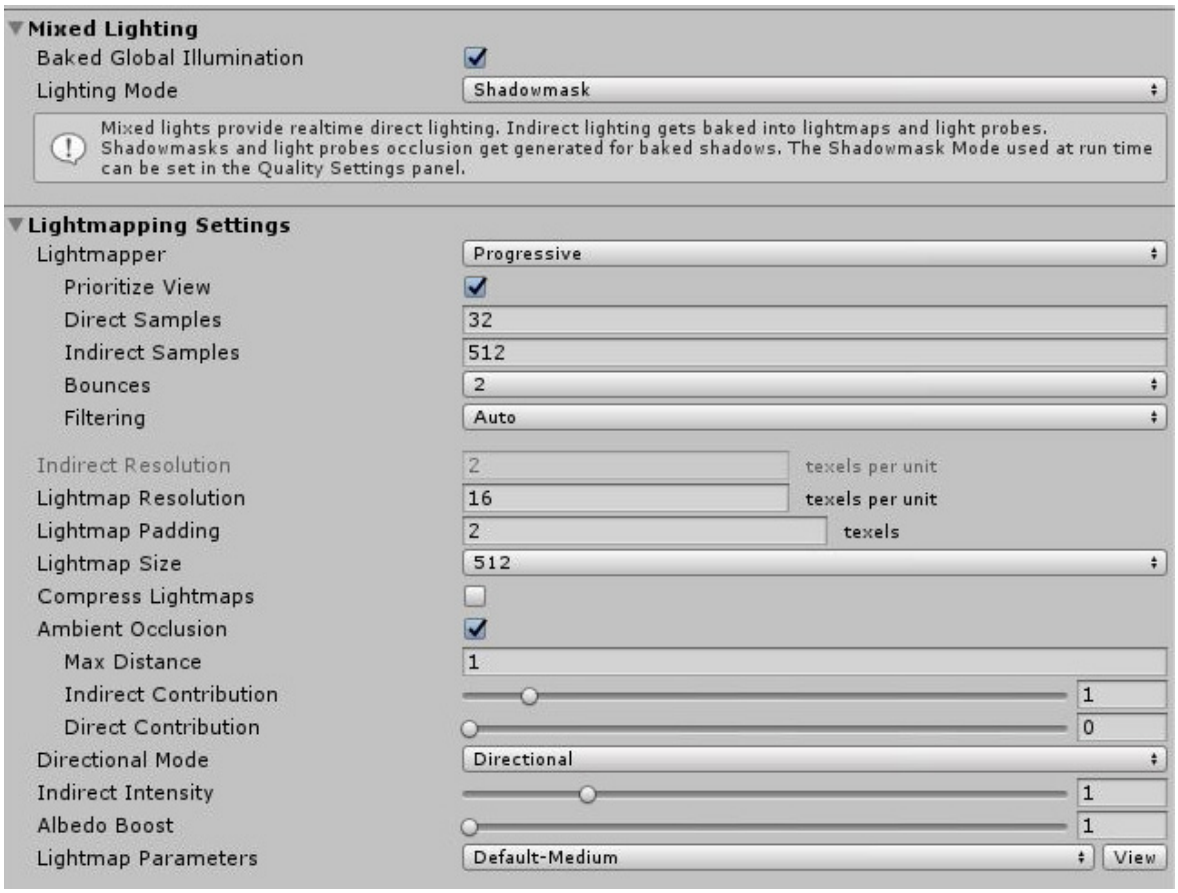
Progressive Lightmapper takes a short preparation step to process geometry and instance updates, and generates the G-buffer and chart masks. It then produces the output immediately and progressively refines it over time for a much-improved interactive lighting workflow. Additionally, baking times are much more predictable because Progressive Lightmapper provides an estimated time while it bakes.

Progressive Lightmapper also bakes global illumination (GI) at the lightmap resolution for each texel individually, without upsampling schemes or relying on any irradiance caches or other global data structures. This makes it robust and allows you to bake selected portions of lightmaps, which makes it faster for you to test and iterate on your **Scene**.

For an in-depth video showing the interactive workflow, see Unity's video walkthrough: In Development - Progressive Lightmapper (YouTube).

Settings

To open the settings, go to **Window** > **Rendering** >**Lighting Settings**.



Property:	Function:
Lighting Mode	Specifies which lighting mode Unity should use for all mixed lights in the Scene. Options are Baked Indirect, Distance Shadowmask, Shadowmask, and Subtractive.
Lightmapper	Use this to specify which internal lighting calculation software to use to calculate lightmaps in the Scene. The options are **Progressive** and **Enlighten**. The default value is **Progressive**. If you want to use the Enlighten system, see documentation on Enlighten.
Prioritize View	Enable this to make the Progressive Lightmapper apply changes to the texels that are currently visible in the **Scene View**, then apply changes to the out-of-view texels.
Direct Samples	The number of samples (paths) shot from each texel. This setting controls the number of samples Progressive Lightmapper uses for direct lighting calculations. Increasing this value can improve the quality of lightmaps, but increases the baking time.
Indirect Samples	The number of samples (paths) shot from each texel. This setting controls the number of samples Progressive Lightmapper uses for indirect lighting calculations. For some Scenes, especially outdoor Scenes, 100 samples should be enough. For indoor Scenes with emissive geometry, increase the value until you see the result you want.
Bounces	Use this value to specify the number of indirect bounces to do when tracing paths. For most Scenes, two bounces is enough. For some indoor Scenes, more bounces might be necessary.
Filtering	Configure the post-processing of lightmaps to limit noise. You can set it to **None**, **Auto** or **Advanced**. The Advanced option offers three additional parameters for manual configuration. In Auto mode, the default values from the Advanced mode are used. For every parameter, Gaussian or A-Trous filter can be used separately. **Auto** — Uses default values for post-processing lightmaps. **Advanced** — Offers three additional parameters for manual configuration. You can use the Gaussian or A-Trous filters separately for direct and indirect settings. Note that the **Gaussian** filter values define the radius, while the **A-Trous** filter value defines the “sigma”. Sigma is a parameter that determines the threshold at which the filter acts on differences in the image.

Property:	Function:			
	Direct Filter	Select a filter to use for direct light in the lightmap.		
		Gaussian	Select this to use a Gaussian filter for direct light in the lightmap.	
			Direct Radius	The radius of the Gaussian filter in texels for direct light in the lightmap. A higher radius increases the blur strength.
		A-Trous	Select this to use an A-Trous filter for direct light in the lightmap.	
			Direct Sigma	The sigma of the A-Trous filter in texels for direct light in the lightmap. A higher sigma increases the blur strength.
		None	Select this to use no filter for direct light in the lightmap.	
	Indirect Filter	Select a filter to use for indirect light in the lightmap.		
		Gaussian	Select this to use a Gaussian filter for indirect light in the lightmap.	
			Indirect Radius	The radius of the Gaussian filter in texels for indirect light in the lightmap. A higher radius increases the blur strength.
		A-Trous	Select this to use an A-Trous filter for indirect light in the lightmap.	
			Indirect Sigma	The sigma of the A-Trous filter in texels for indirect light in the lightmap. A higher sigma increases the blur strength.
	Ambient Occlusion Filter	Select a filter to use for Ambient Occlusion (see below) in the lightmap. Filter only available when you enable **Ambient Occlusion**.		
		Gaussian	Select this to use a Gaussian filter for Ambient Occlusion in the lightmap.	
			Ambient Occlusion Radius	The radius of the Gaussian filter in texels for Ambient Occlusion in the lightmap. A higher radius increases the blur strength.
		A-Trous	Select this to use an A-Trous filter for Ambient Occlusion in the lightmap.	
			Ambient Occlusion Sigma	The sigma of the A-Trous filter in texels for Ambient Occlusion in the lightmap. A higher sigma increases the blur strength.
	None	Select this to use no filter for indirect light in the lightmap.		
	Lightmap Resolution	Use this to specify the number of texels per unit to use for lightmaps. Increasing this value improves lightmap quality, but also increases bake times. Note that doubling this value causes the number of texels to quadruple (because the value refers to both the height and width of the lightmap). Check the **Occupied texels** count in the stats, documented below this table.		
	Lightmap Padding	Use this to specify the separation (in texel units) between separate shapes in the baked lightmap. The default value is 2.		
	Lightmap Size	The size (in pixels) of the full lightmap texture, which incorporates separate regions for the individual **GameObject** textures. The default value is 1024.		

Property:	Function:		
	Compress Lightmaps	A compressed lightmap requires less storage space, but the **compression** process can introduce unwanted visual effects into the texture. Tick this checkbox to compress lightmaps, or untick it to keep them uncompressed. The checkbox is ticked by default.	
	Ambient Occlusion	Tick this checkbox to open a group of settings which allow you to control the relative brightness of surfaces in ambient occlusion. Higher values indicate a greater contrast between the occluded and fully lit areas. This only applies to the indirect lighting calculated by the GI system. This setting is enabled by default.	
		Max Distance	Set a value to control how far the lighting system casts rays in order to determine whether or not to apply occlusion to an object. A larger value produces longer rays and contributes more shadows to the lightmap, while a smaller value produces shorter rays that contribute shadows only when objects are very close to one another. A value of 0 casts an infinitely long ray that has no maximum distance. The default value is 1.
		Indirect Contribution	Use the slider to scale the brightness of indirect light as seen in the final lightmap (that is, ambient light, or light bounced and emitted from objects) from a value between 0 and 10. The default value is 1. Values less than 1 reduce the intensity, while values greater than 1 increase it.
		Direct Contribution	Use the slider to scale the brightness of direct light from a value between 0 and 10. The default value is 0. The higher this value is, the greater the contrast applied to the direct lighting.
	Directional Mode	You can set the lightmap up to store information about the dominant incoming light at each point on the objects' surfaces. See documentation on Directional Lightmapping for further details. The default mode is **Directional**.	
		Directional	In **Directional** mode, Unity generates a second lightmap to store the dominant direction of incoming light. This allows diffuse normal mapped materials to work with the GI. Directional mode requires about twice as much storage space for the additional lightmap data. Directional lightmaps cannot be decoded on SM2.0 hardware or when using GLES2.0. They fall back to Non-Directional lightmaps.
		Non-directional	**Non-directional** mode switches Directional option off.
	Indirect Intensity	Use this slider to control the brightness of indirect light stored in realtime and baked lightmaps, from a value between 0 and 5. A value above 1 increases the intensity of indirect light while a value of less that 1 reduces indirect light intensity. The default value is 1.	
	Albedo Boost	Use this slider to control the amount of light Unity bounces between surfaces, from a value between 1 and 10. To do this, Unity intensifies the albedo of materials in the Scene. Increasing this draws the albedo value towards white for indirect light computation. The default value of 1 is physically accurate.	
	Lightmap Parameters	Unity uses a set of general parameters for the lightmapping in addition to properties of the Lighting window. A few defaults are available from the menu for this property but you can also create your own lightmap parameter file using the **Create New** option. See the Lightmap Parameters page for further details. The default value is **Default-Medium**.	

Statistics

The panel below the **Auto Generate** and **Generate Lighting** options shows statistics about the lightmapping, including:

The number of lightmaps that Unity has created

Memory Usage: The amount of memory required for the current lightmapping.

Occupied Texels: The number of texels that are occupied in lightmap UV space.

Lightmaps in view: The number of lightmaps in the Scene view.

Lightmaps not in view: The number of lightmaps that are out of view.

Converged: All calculations for these lightmaps are complete.

Not Converged: Baking is still in progress for these lightmaps.

Bake Performance: The number of mrays per second. If this is low (that is, less than 2) you should adjust your settings or your hardware to process more light rays at a time.

In **Auto** mode, Unity automatically calculates the lightmaps and Light Probes. If you have **Auto** disabled, you need to press the **Build** button to start the bake.

During baking

Progressive Lightmapper provides options to monitor and stop the bake while it is in progress, if you need to.

ETA

The progress bar that appears while Unity is baking the lightmap provides an “estimated time of arrival” (displayed as **ETA**). This is the estimated time in seconds for the current bake to complete. This allows for much more predictable baking times, and allows you to quickly learn how much time baking takes with your current lighting settings.

Force Stop

During manual baking, press **Force Stop** at any time to halt the baking process. This allows you to stop the process as soon as you see results that look good.

Progressive Lightmapper added in 2018.1

2018–03–28 Page amended with limited editorial review

Lightmapping: Getting started

Leave feedback

This page provides an introduction to lightmapping in Unity. Lightmapping is the process of pre-calculating the brightness of surfaces in a Scene, and storing the result in a chart or “light map” for later use.

Unity uses a system called the Progressive Lightmapper, which bakes **lightmaps** for your Scene based on how your Scene is set up within Unity, taking into account Meshes, Materials, Textures, and Lights. Lightmapping is an integral part of the **rendering** engine; when your **lightmaps** are created, **GameObjects** automatically use them.

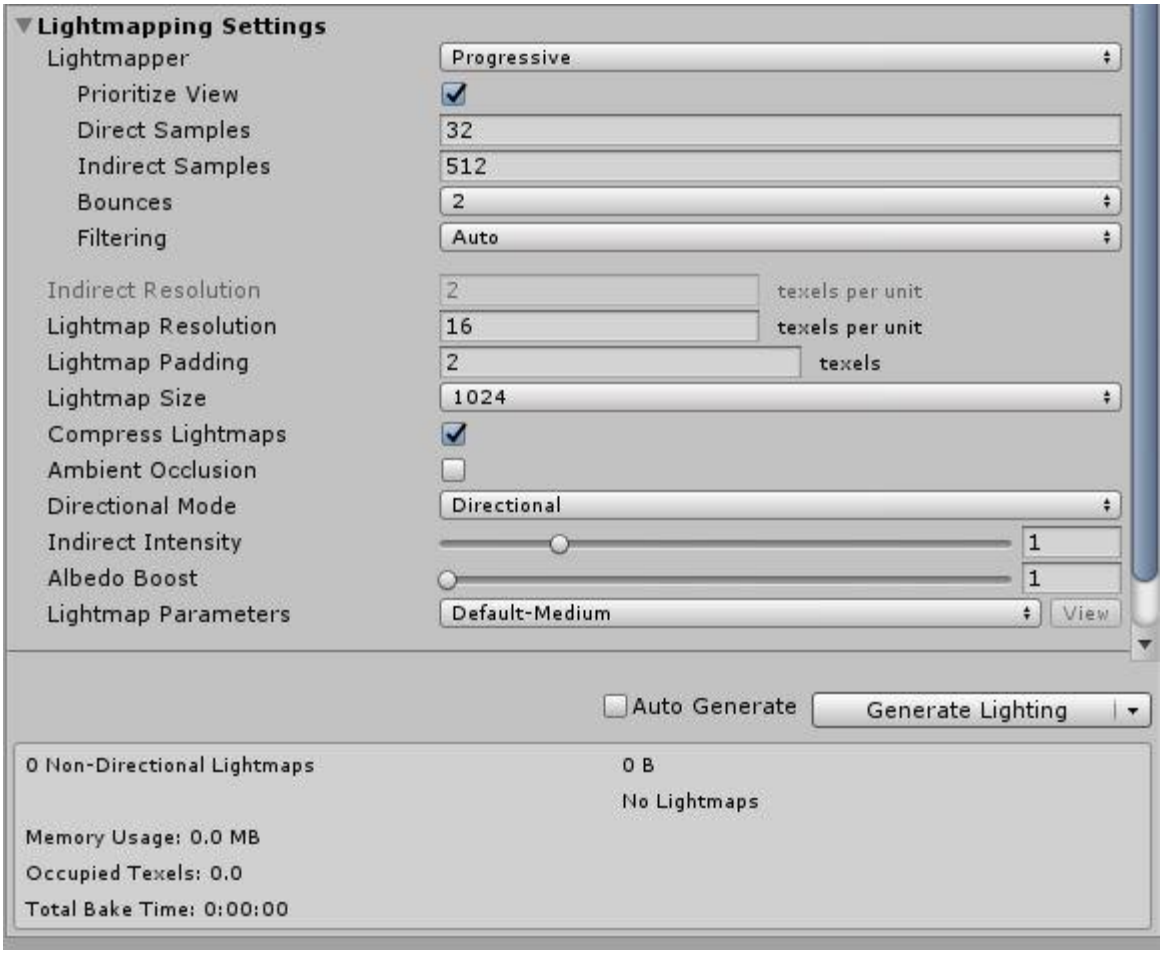
For information about specific lightmapping-related settings, see documentation on Global Illumination.

Preparing the Scene and baking the lightmaps

Select **Window** > **Lighting** > **Settings** from the Unity Editor menu to open the Lighting window. Make sure any Mesh you want to apply a light map to has proper UVs for lightmapping. The easiest way to do this is to open the Mesh import settings and enable the **Generate Lightmap UVs** setting.

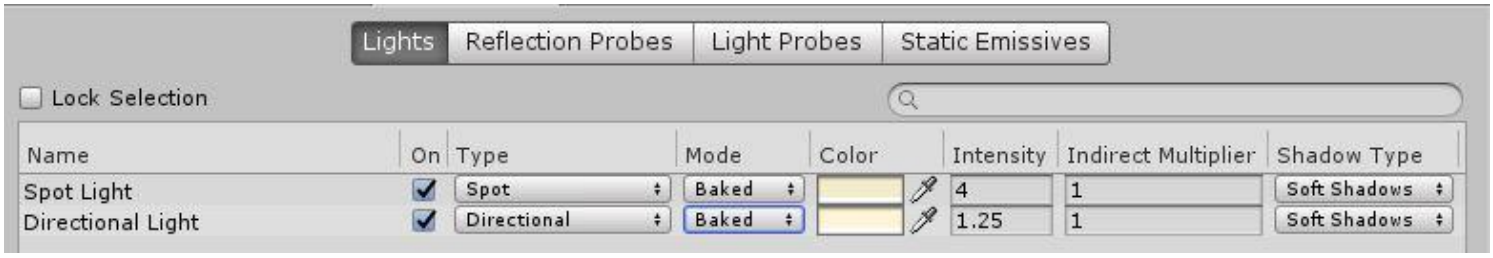
Next, to control the resolution of the lightmaps, go to the **Lightmapping Settings** section and adjust the **Lightmap Resolution** value.

Note: To have a better understanding of how you spend your lightmap texels, look at the small **Shaded debug scene visualization mode within the** Scene View**, switch to** Baked Lightmap__ and tick the **Show Resolution** checkbox.



In the **Mesh Renderer** and **Terrain** components of your GameObjects, enable the **Lightmap Static** property. This tells Unity that those GameObjects don't move or change, so Unity can add them to a lightmap. In the **Mesh Renderer component**, you can also use the **Scale In Lightmap** parameter to adjust the resolution of your lightmap's static **Mesh** or **Terrain**.

You can also adjust settings for Lights in the Light Explorer.



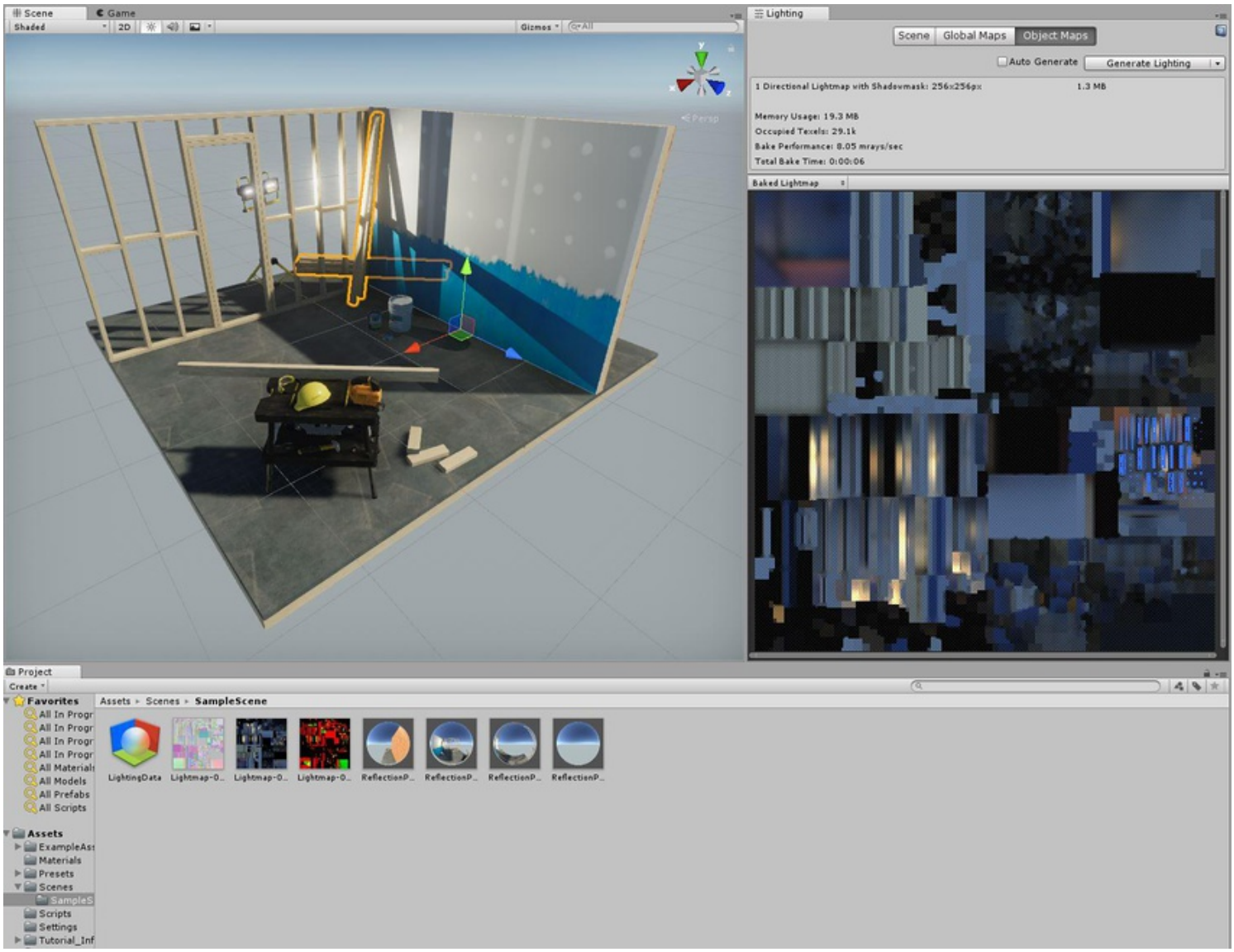
To generate lightmaps for your Scene:

At the bottom of the **Scene** tab on the Lighting window, click **Generate Lighting** (or ensure that **Auto Generate** is ticked).

A progress bar appears in Unity Editor's status bar, in the bottom-right corner.

When baking is complete, you can see all the baked lightmaps in the **Global Maps** and **Object Maps** tabs of the Lighting window.

When lightmapping is complete, Unity's Scene and Game views update automatically.



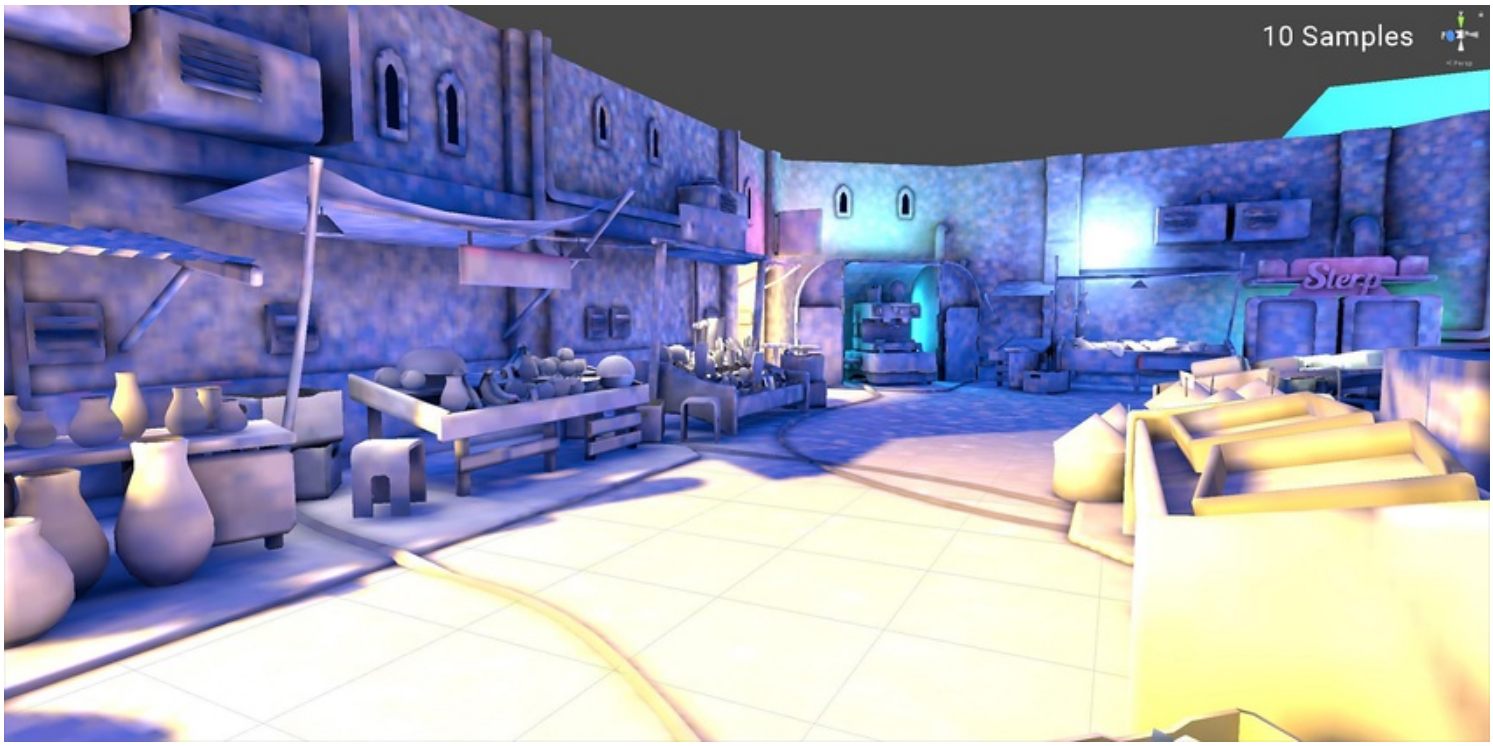
To see the UV chart of the Mesh, click on a GameObject that has **Lightmap Static** enabled, then navigate to the Inspector window and select the **Object Maps** tab. Here, you can switch between different light map visualization modes. When you manually generate lighting, Unity adds Lighting Data Assets, baked lightmaps and Reflection Probes to the Assets folder.

Tweaking bake settings

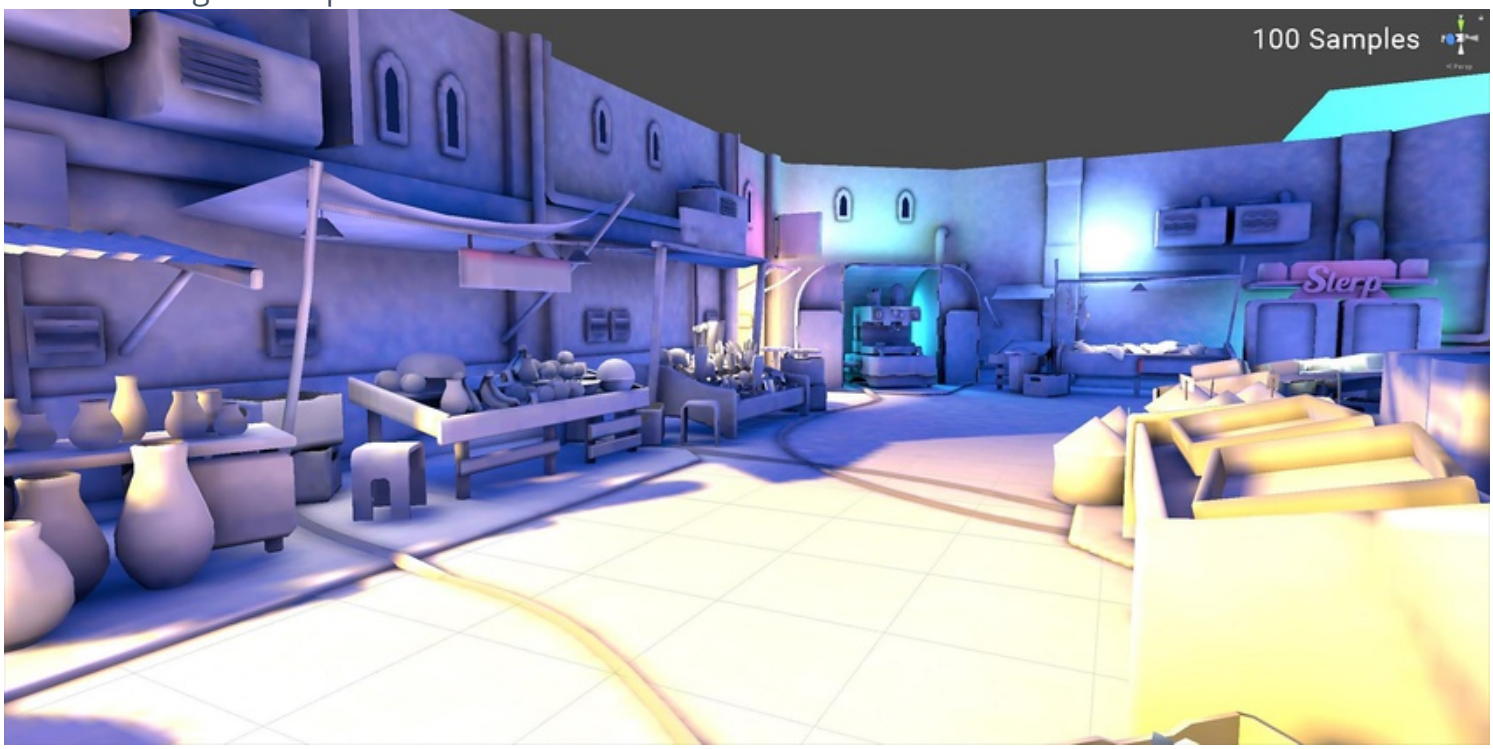
The final look of your Scene depends on your lighting set-up and bake settings. Let's take a look at an example of some basic settings that can improve lighting quality.

Sample count Progressive Lightmapper generates color values resulting from a single ray in order to remove noise. These color values are called samples. There are two settings that control the number of samples the Progressive Lightmapper uses for direct and indirect lighting calculations: **Direct Samples** and **Indirect Samples**. To find these, open the Lighting window (**Window** > **Lighting** > **Settings**), and go to **Lightmapping Settings** > **Lightmapper**.

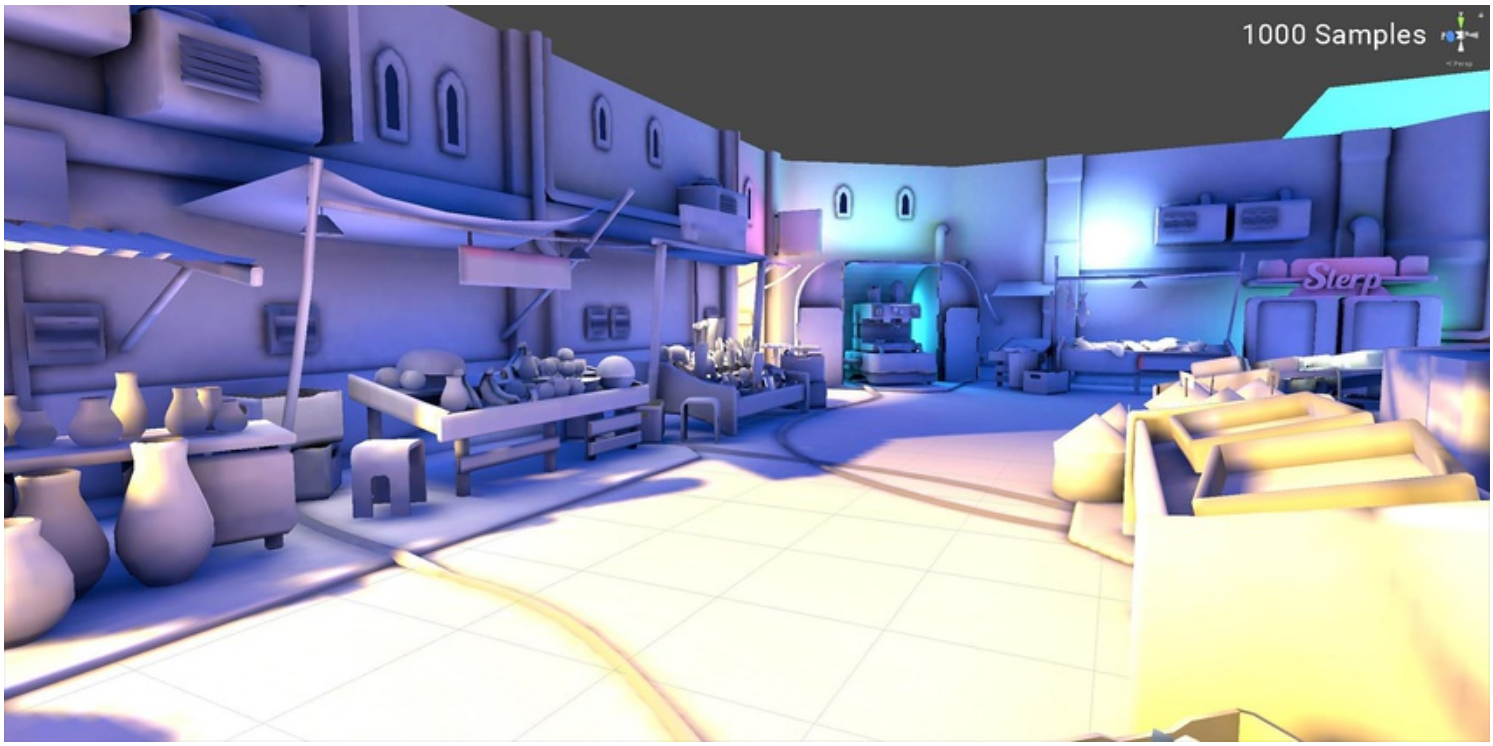
Higher sample values reduce noise and can improve the quality of the lightmaps, but they also increase the bake time. The images below shows how a higher number of samples increases the quality of lightmaps without using filtering, and produces results that are less noisy.



A Scene using 10 samples



A Scene using 100 samples



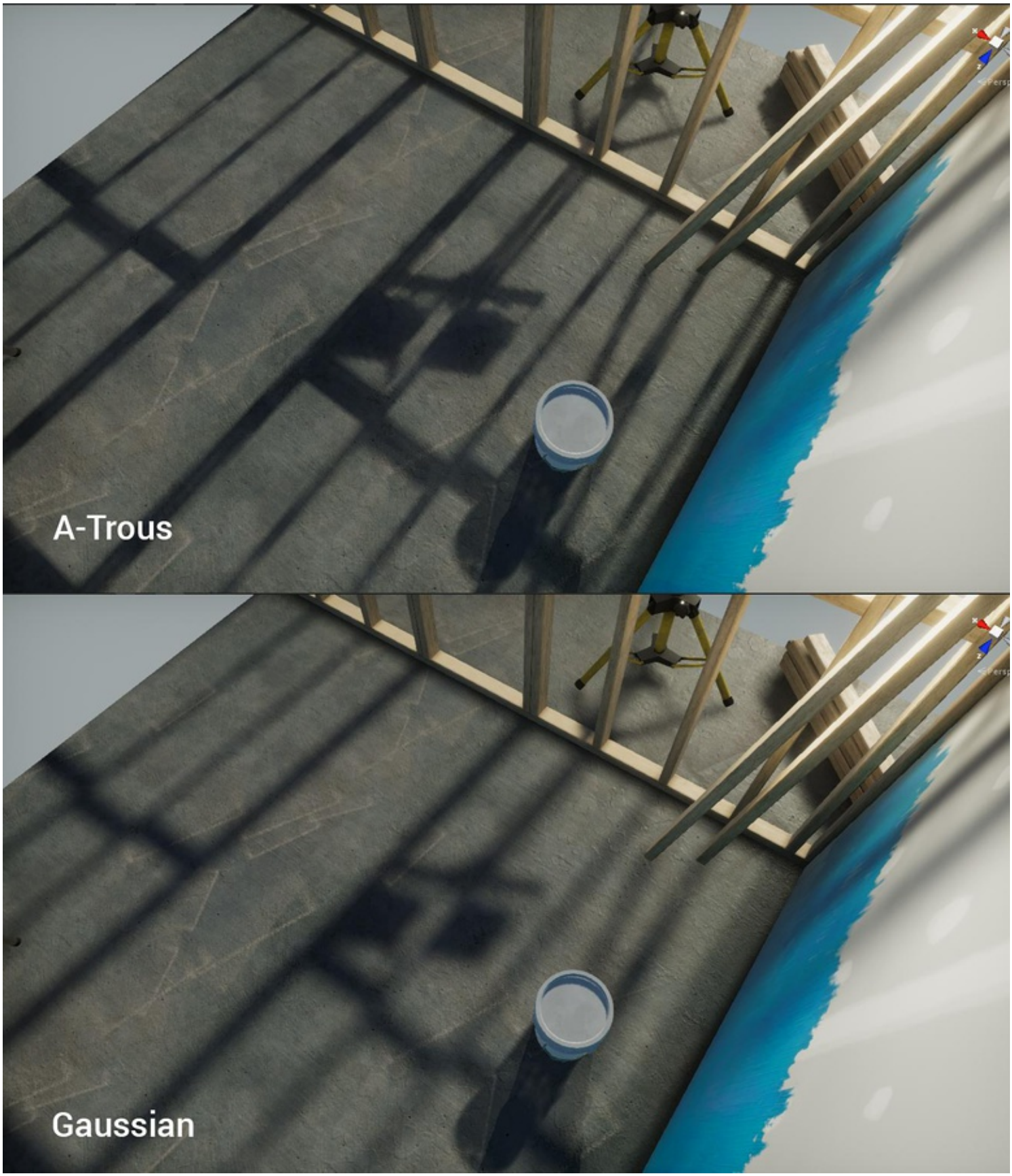
A Scene using 1000 samples

Environment Lighting

In addition to all Light sources, Environment Lighting can also contribute to Global Illumination. You can assign a custom **Skybox** Material instead of default **Procedural Skybox** and adjust intensity. The following image shows how lighting can change in the Scene with and without **Environment Lighting**, and provide softer results. The settings for light sources are the same in both Scenes. Unity provides several custom **HDRI** assets in the Asset Store.

Filtering

Filtering allows you to blur noisy results. The Progressive Lightmapper offers two different types of filtering: **Gaussian** and **A-Trous**. When you enable **Advanced** settings, you can apply these filters for **Direct**, **Indirect** and **Ambient Occlusion** separately. For more information, see documentation on Progressive Lightmapper.



2018–03–28 Page amended with limited editorial review

Progressive Lightmapper added in 2018.1

Lightmap seam stitching

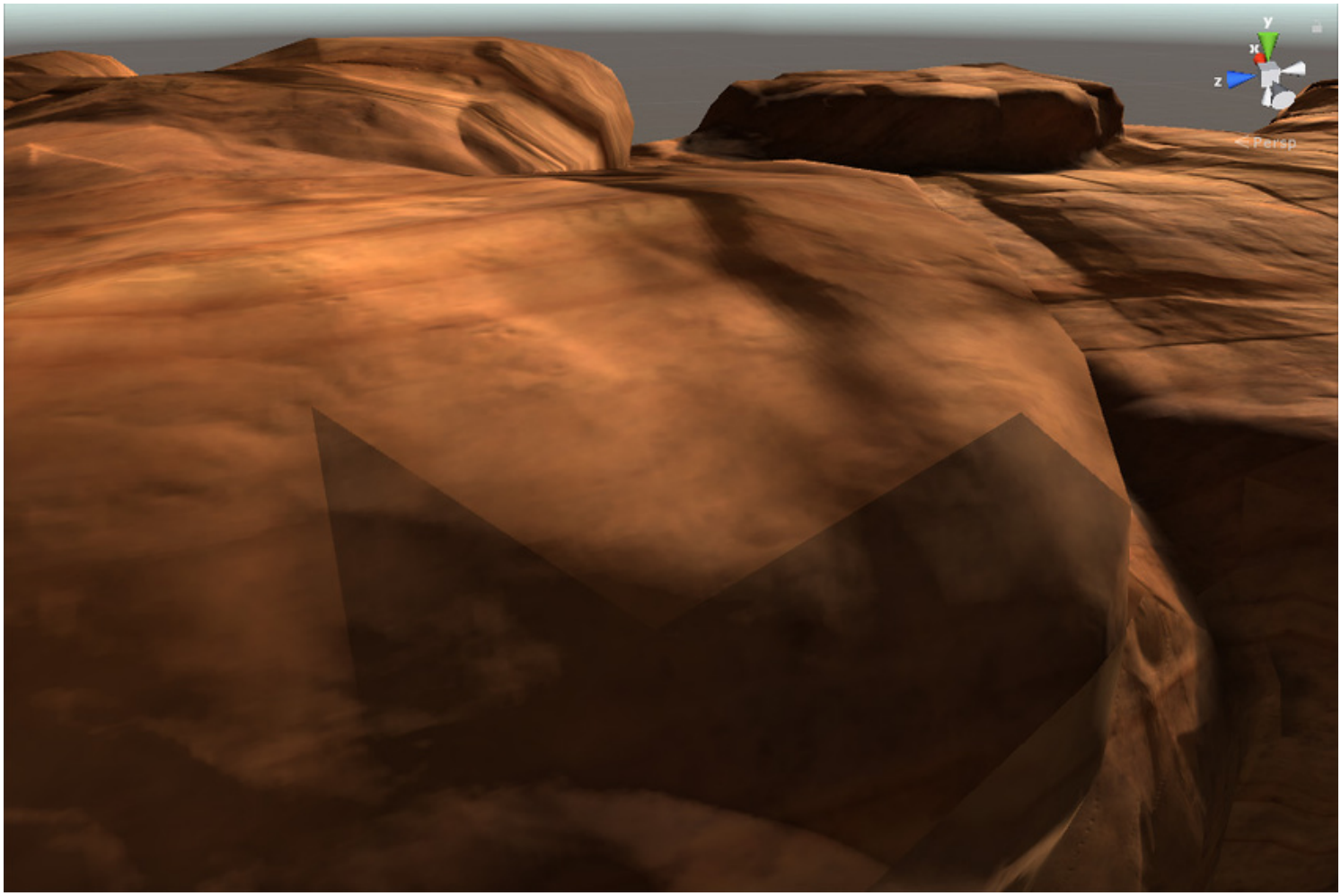
Leave feedback

Lightmap seam stitching is a technique that smooths unwanted hard edges in **GameObjects** rendered with baked **lightmaps**.

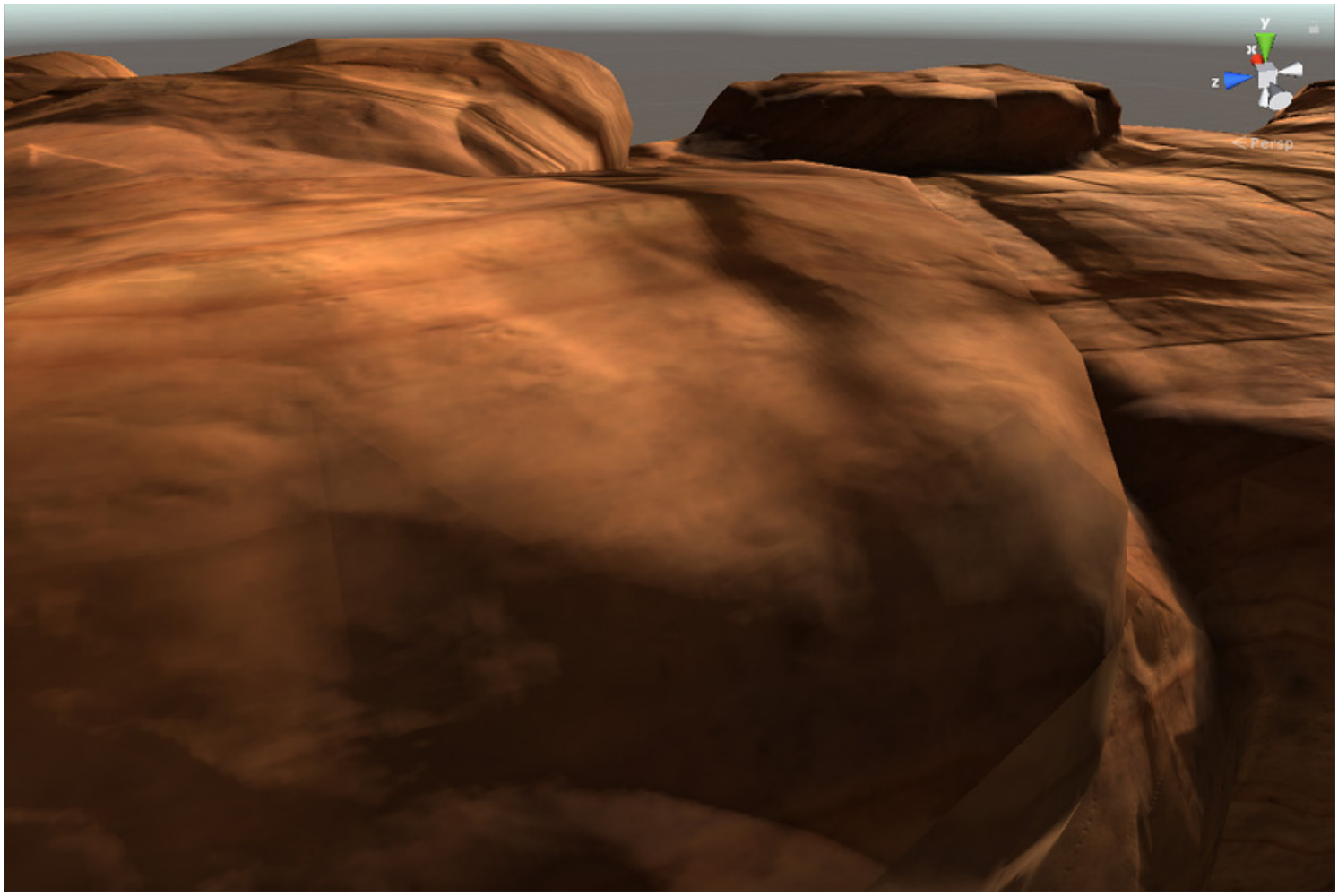
Seam stitching works with the Progressive Lightmapper for lightmap baking. Seam stitching only works on single GameObjects; multiple GameObjects cannot be smoothly stitched together.

Lightmapping involves Unity unwrapping 3D GameObjects onto a flat lightmap. Unity identifies **mesh** faces that are close together but separate from each other as being separate in lightmap space; the edges of these meshes are called “seams”. Seams are ideally invisible but they can sometimes appear to have hard edges depending on the light. This is because the GPU cannot blend texel values between charts that are separated in the lightmap.

Seam stitching is a technique that fixes these issues. When you enable seam stitching, Unity does extra computations to amend the lightmap to improve each seam’s appearance. Stitching is not perfect, but it often improves the final result substantially. Seam stitching takes extra time during baking due to extra calculations Unity makes, so Unity disables it by default. You enable Stitching on the GameObject’s MeshRenderer.

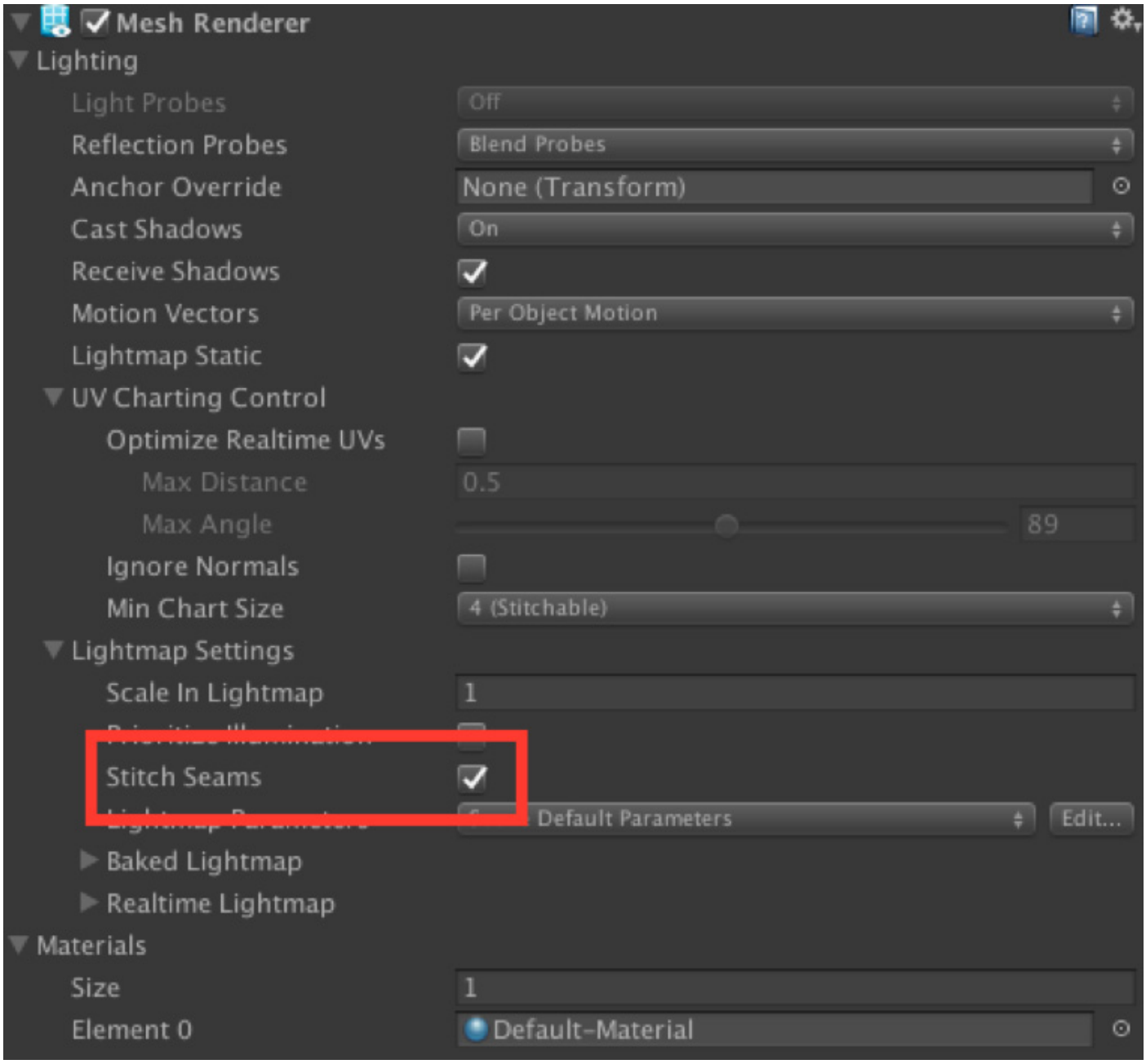


A Scene without seam stitching



A Scene with seam stitching

To enable seam stitching on a GameObject, go to the GameObject's **Mesh Renderer component**, open the **Lightmap Settings** section (only accessible if you are using the Progressive Lightmapper), and tick **Stitch Seams**.



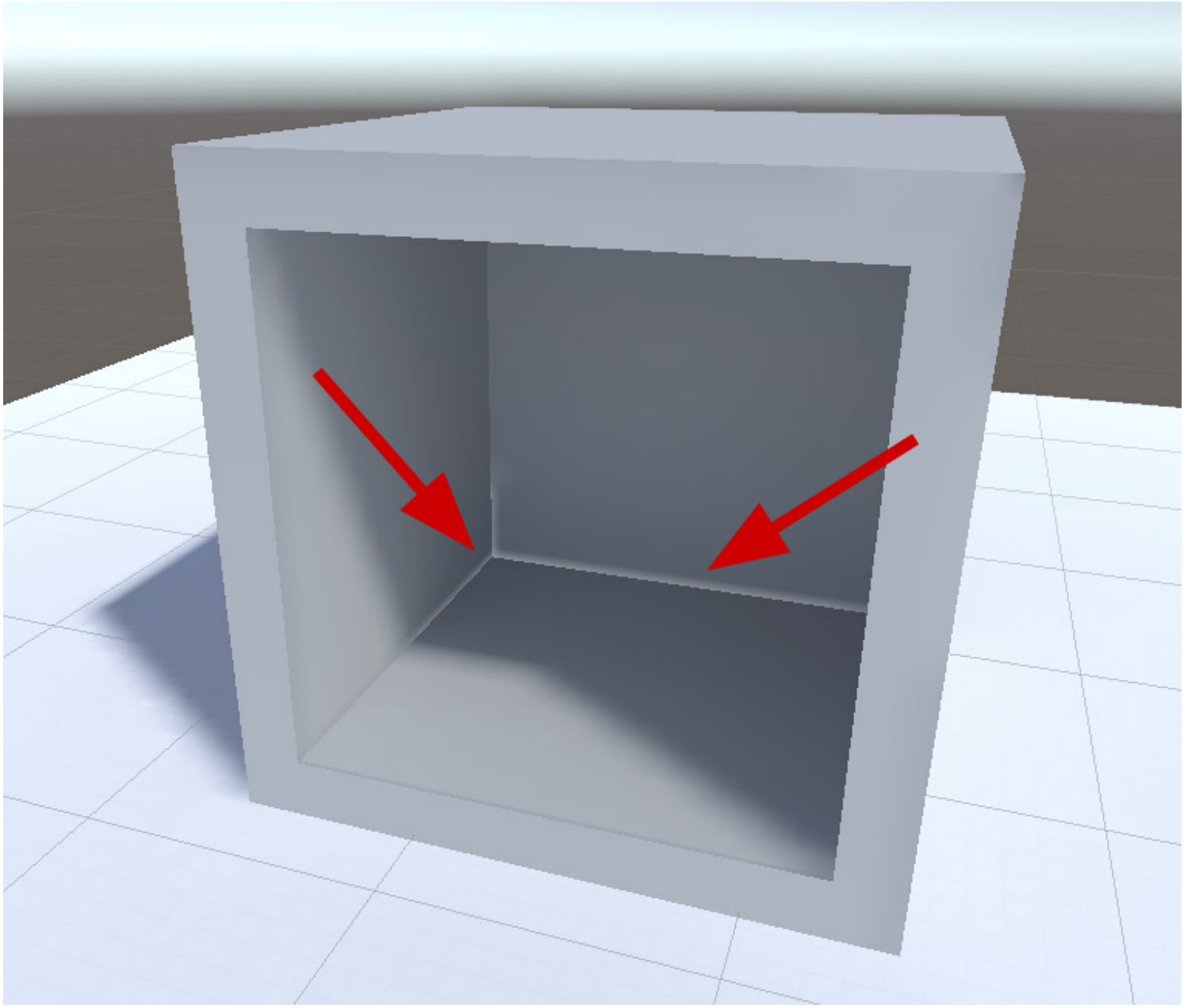
2017–09–04 Page published with limited editorial review

Seam stitching added in 2017.2

UV overlap feedback

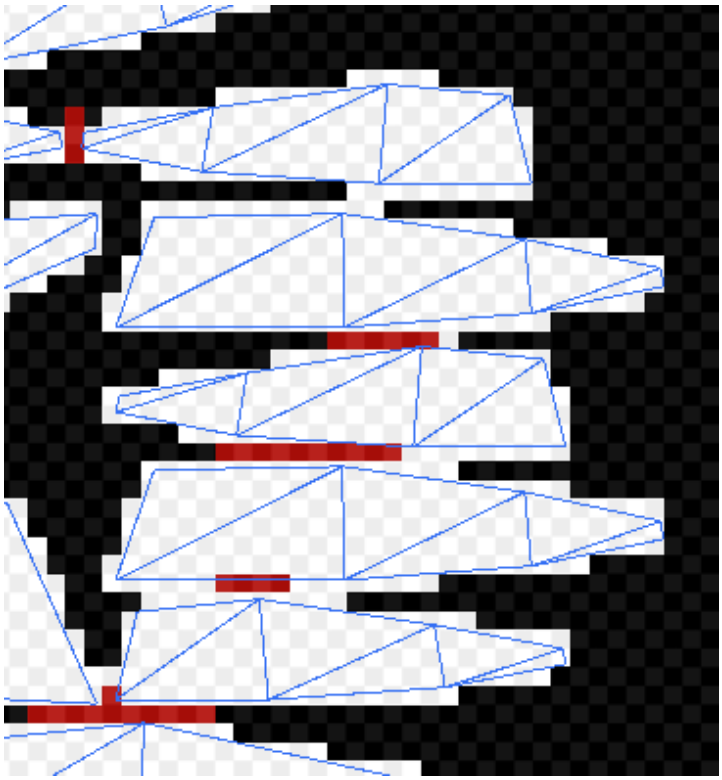
Leave feedback

Each **lightmap** contains a number of **charts**. At run time, Unity maps these charts onto **mesh** faces, and uses the charts' lighting data to calculate the final appearance. Because of the way GPU sampling works, data from one chart can bleed onto another if they are too close to each other. This usually leads to unintended aliasing, pixelation, and other graphical results (these are called artifacts).



Example of graphical artifacts due to chart bleeding

To avoid light bleeding, there must be a sufficient amount of space between charts. When a GPU samples a lightmap, the lighting system calculates the final sample value from the four texels closest to the sampled point (assuming bilinear filtering is used). These four texels are called the bilinear "neighborhood" of the sampled point. Charts are too close together if they overlap - that is, if the neighbourhood of any point inside a chart overlaps with the neighborhood of any point in another chart. In the image below, the white **pixels** indicate chart neighbourhoods, and red **pixels** indicate overlapping neighbourhoods.



Red pixels indicate overlapping chart neighbourhoods

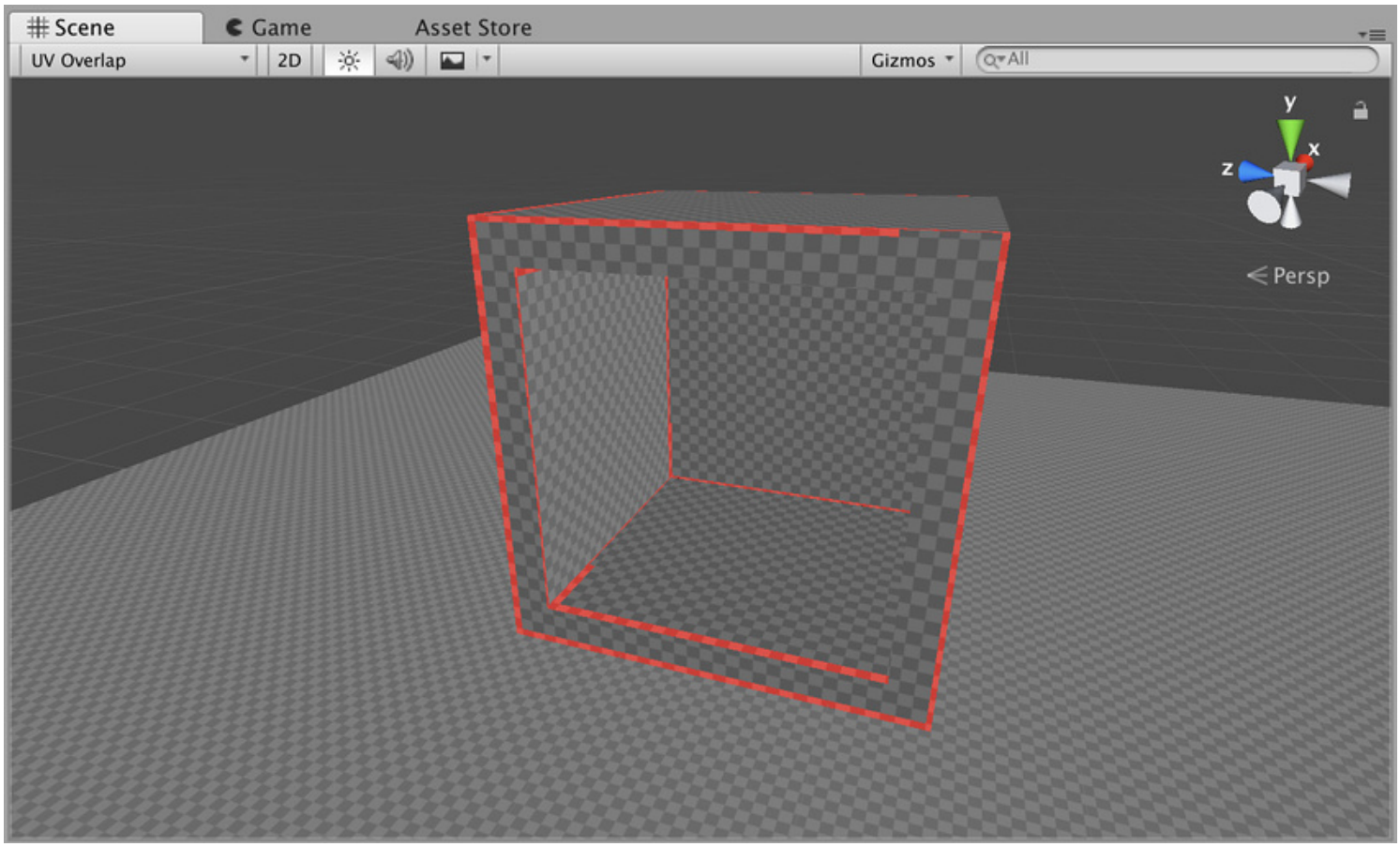
Determining optimal chart placements and spacing can be difficult, because it depends on several parameters (such as lightmap resolution, mesh UVs, and Importer settings). For this reason, Unity provides the ability to identify these issues easily, as outlined in the following section.

Identification

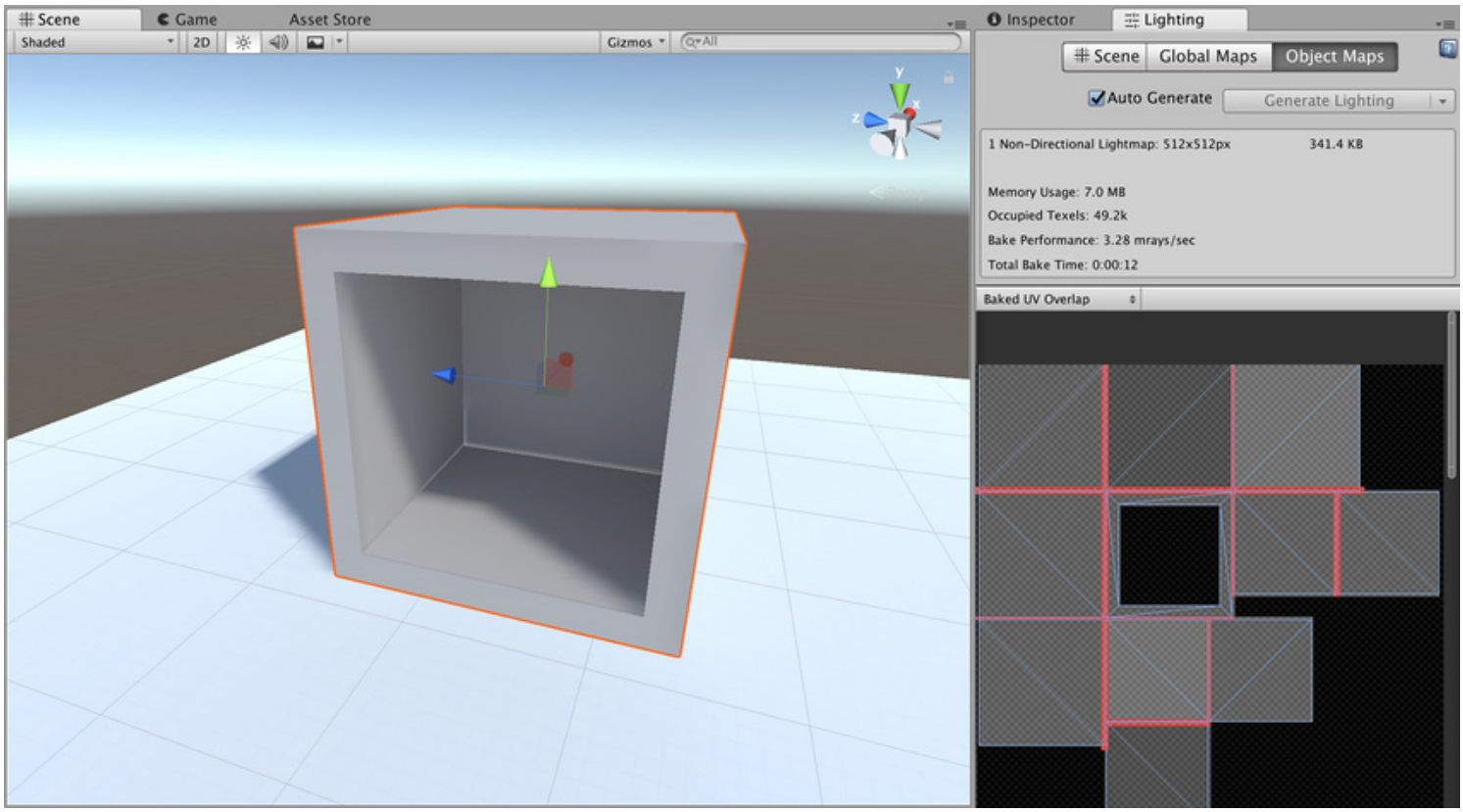
There are three ways to identity overlaps:

Keep an eye on Unity's console. If Unity detects overlapping UVs, it prints a warning message with a list of affected **GameObjects**.

Use the **UV Overlap** draw mode in the **Scene View** (see GI visualizations in the Scene View for more information). When you have this mode enabled, Unity adds a red highlight to chart texels that are too close to texels of other charts. This is especially useful if you discover an artefact in the **Scene** view, and want to quickly examine whether UV overlap is causing it.



Scene View using UV Overlap draw mode (see dropdown in top left)
Use Object Maps in the Lighting tab. Select an object and go to the Lighting tab and then choose Object Maps. Make sure to select Baked UV Overlap in the dropdown. Problematic texels are colored red in this view.



Object Maps in the Lighting tab

Solutions

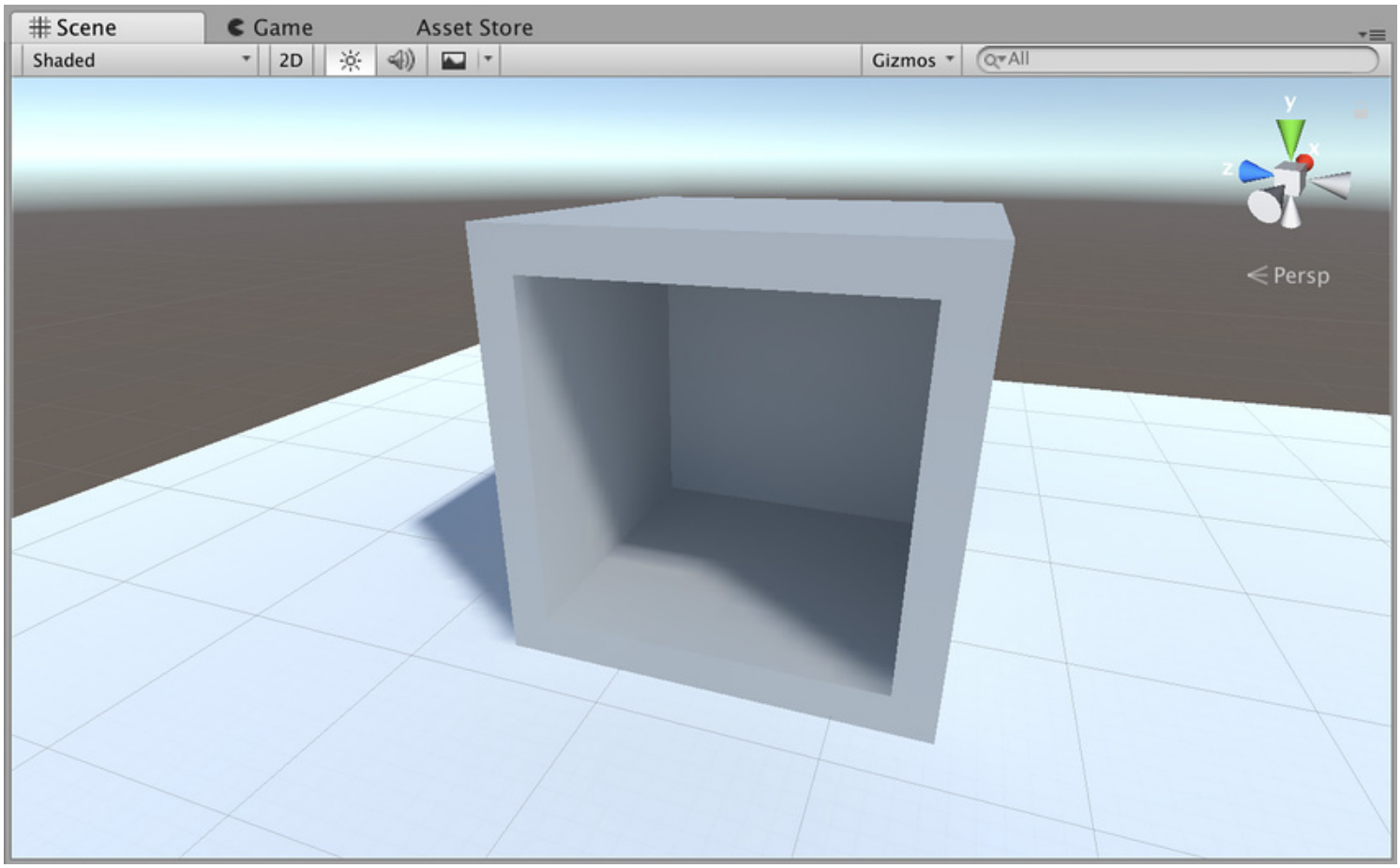
There is no one single solution for UV overlap, because there are so many things that can cause it. Here are the most common solutions to consider:

If Unity is automatically creating the lightmap UVs, you can increase the **Pack Margin**. To do this, navigate to the Model tab of the Mesh's import settings. Make sure **Generate Lightmap UVs** is enabled, then fold out **Advanced** and use the **Pack Margin** slider to increase the value. This creates more spacing between charts, which reduces likelihood of overlap. However, this also increase the total space requirement for the lightmap, so try to apply enough spacing to avoid artifacts, but no more. For more information on lightmap UVs that Unity creates automatically, see documentation on Generating lightmapping UVs.

If you provide lightmap UVs yourself, you can try adding margins using your modelling package.

Increase the resolution of the entire lightmap. This will increase the numbers of pixels between the charts, and therefore reduce the likelihood of bleeding. The downside is that your lightmap may become too large. You can do this in the **Lighting** tab under **Lightmapper Settings**.

Increase the resolution of a single GameObject. This allows you to increase lightmap resolution only for GameObjects that have overlapping UVs. Though less likely, this can also increase your lightmap size. You can change a GameObject's lightmap resolution inside its **Mesh Renderer** under **Lightmap Settings**.



Same mesh as before, but without bleeding artifacts

Progressive Lightmapper added in 2018.1

2018–03–28 Page published with limited editorial review

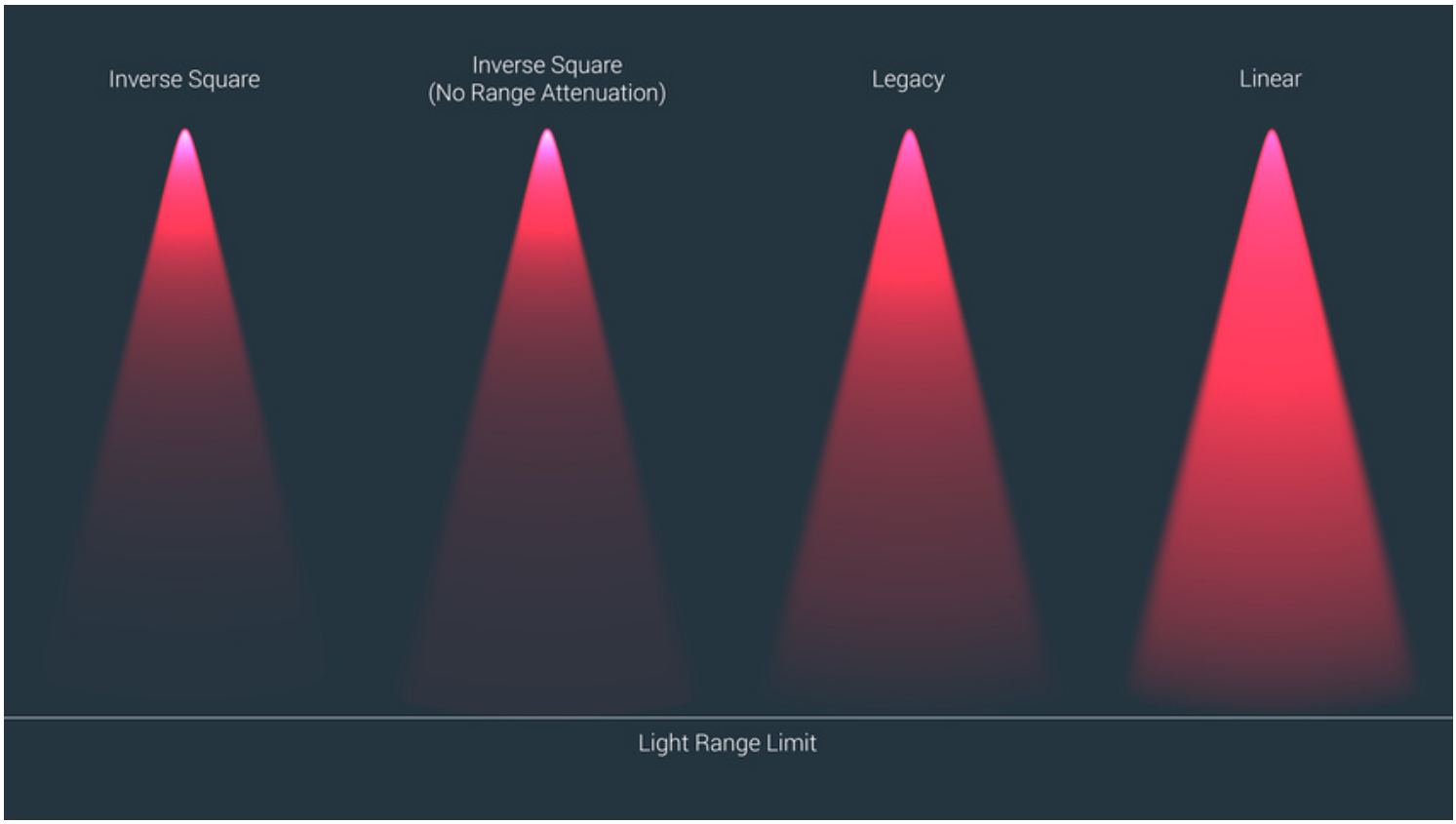
Custom fall-off

Leave feedback

In the real world, light fades over distance, and dim lights have a lower range than bright lights. The term “fall-off” refers to the rate at which light fades. Alongside Unity’s default fall-off lighting behaviour, you can also use custom fall-off settings.

Progressive **Lightmapper** provides custom fall-off presets, which you can implement via script. See the image below the table for a visual representation of how these work, and the code sample below the image for an example of how to use this functionality.

Property	Function
`InverseSquared`	Apply an inverse-squared distance fall-off model. This means the light intensity decreases inversely proportional to the square of location’s distance to the light source. For more information , see Wikipedia: Inverse-square law. This option is the most physically accurate.
`InverseSquaredNoRangeAttenuation`	Apply an inverse-squared distance fall-off model with no smooth range attenuation. This works in the same way as `InverseSquared`, but the lighting system does not take into account the attenuation for the range parameter of punctual lights (that is, point lights and spotlights).
`Legacy`	Apply a quadratic fall-off model. This model bases the light attenuation on the range of the light source. The intensity diminishes as the light gets further away from the source, but there is a very sharp and unnatural drop in the attenuation, and the visual effect is not realistic.
`Linear`	Apply a linear fall-off model. In this model, attenuation is inversely proportional to the distance from the light, and the fall-off diminishes at a fixed rate from its source.



An example of the visual effect of each custom fall-off preset

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Experimental.GlobalIllumination;
using UnityEngine.SceneManagement;

[ExecuteInEditMode]
public class ExtractFalloff : MonoBehaviour
{
    public void OnEnable()
    {
        Lightmapping.RequestLightsDelegate testDel = (Light[] requests, Unity.Co
        {
            DirectionalLight dLight = new DirectionalLight();
            PointLight point = new PointLight();
            SpotLight spot = new SpotLight();
            RectangleLight rect = new RectangleLight();
            LightDataGI ld = new LightDataGI();

            for (int i = 0; i < requests.Length; i++)
            {
                Light l = requests[i];
                switch (l.type)
                {
```

```
                    case UnityEngine.LightType.Directional: LightmapperUtils.Ext
                    case UnityEngine.LightType.Point: LightmapperUtils.Extract(l
                    case UnityEngine.LightType.Spot: LightmapperUtils.Extract(l,
                    case UnityEngine.LightType.Area: LightmapperUtils.Extract(l,
                    default: ld.InitNoBake(l.GetInstanceID()); break;
                }

            ld.falloff = FalloffType.InverseSquared;
                lightsOutput[i] = ld;
            }
        };

        Lightmapping.SetDelegate(testDel);
    }

    void OnDisable()
    {
        Lightmapping.ResetDelegate();
    }
}
```

Progressive Lightmapper added in 2018.1

2018–03–28 Page published with limited editorial review

Lightmap Parameters

Leave feedback

SWITCH TO SCRIPTING

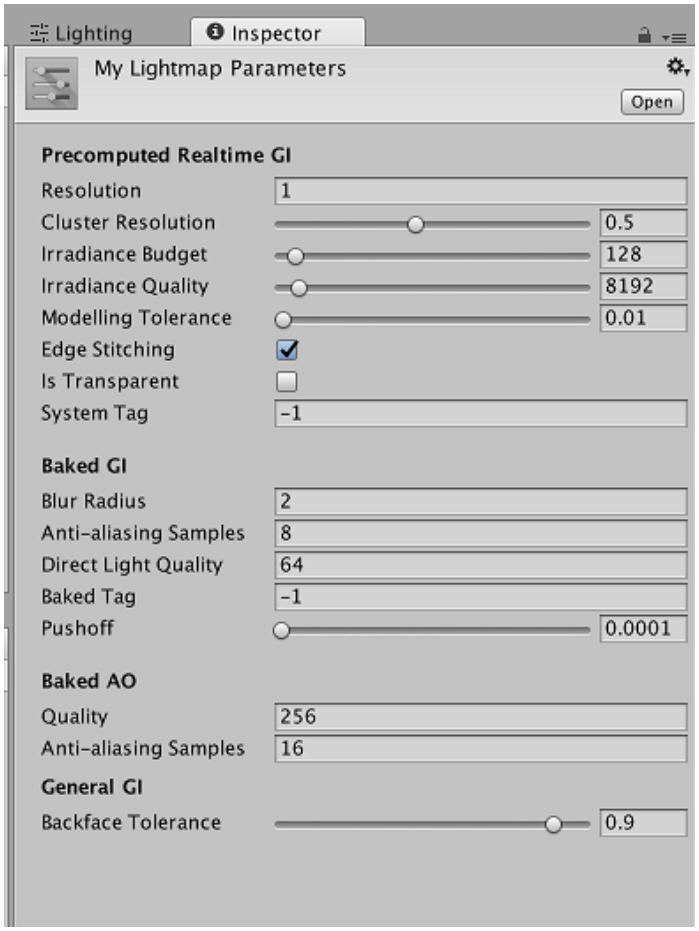
A **Lightmap** Parameters **Asset** stores a set of values for the parameters which control Unity's Global Illumination (GI) features. These Assets allow you to define and save different sets of values for lighting, for use in different situations.

To create a new Lightmap Parameters Asset, right-click in the Project window and go to **Create** > **New Parameters Asset**. Unity stores this in your Project folder.

A Lightmap Parameters Asset called **New LightmapParameters**, shown in the Project window

Lightmap Parameters Assets allow you to quickly create presets optimized for different types of **GameObjects**, or for different platforms and different Scene types (for example, indoor or outdoor Scenes).

When you click on a Lightmap Parameters Asset in the Project window, the **Inspector** window displays the values defined in that Asset. The parameters and their descriptions are listed in the table below.



Precomputed Realtime GI

Property	Function

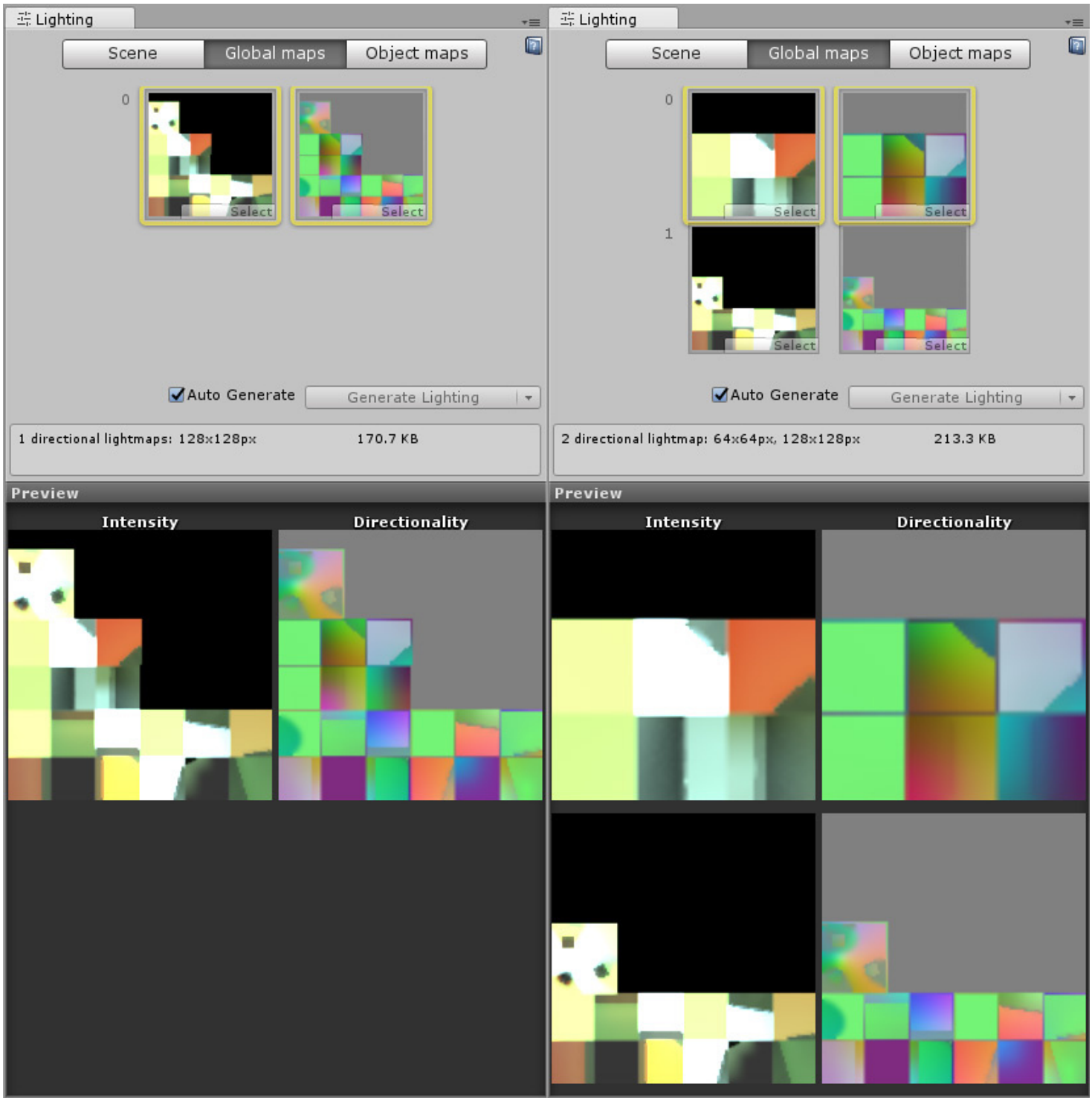
Property	Function
Resolution	This value scales the **Realtime Resolution** value in the Scene tab of the Lighting Window (menu: **Window** > **Rendering** > **Lighting Settings** > **Scene**) to give the final resolution of the lightmap in texels per unit of distance.
Cluster Resolution	The ratio of the cluster resolution (the resolution at which the light bounces are calculated internally) to the final lightmap resolution. See documentation on GI Visualizations in the Scene view for more information.
Irradiance Budget	This value determines the precision of the incoming light data used to light each texel in the lightmap. Each texel's lighting is obtained by sampling a "view" of the Scene from the texel's position. Lower values of irradiance budget result in a more blurred sample. Higher values increase the sharpness of the sample. A higher irradiance budget improves the lighting, but this increases run-time memory usage and might increase CPU usage.
Irradiance Quality	Use the slider to define the number of rays that are cast and used to compute which clusters affect a given output lightmap texel. Higher values offer visual improvements in the lightmap, but increase precomputing time in the Unity Editor. The value does not affect runtime performance.
Modelling Tolerance	This value controls the minimum size of gaps in **Mesh** geometry that allows light to pass through. Make this value lower to allow light to pass through smaller gaps in your environment.
Edge Stitching	If enabled, this property indicates that UV charts in the lightmap should be joined together seamlessly, to avoid unwanted visual artifacts.
Is Transparent	If enabled, the object appears transparent during the Global Illumination calculations. Back-faces do not contribute to these calculations, and light travels through the surface. This is useful for invisible emissive surfaces.
System Tag	A group of objects whose lightmap Textures are combined in the same lightmap atlas is known as a "system". The Unity Editor automatically defines additional systems and their accompanying atlases if all the objects can't be fitted into a single atlas. However, it is sometimes useful to define separate systems yourself (for example, to ensure that objects inside different rooms are grouped into one system per room). Change the **System Tag** number to force new system and lightmap creation. The exact numeric sequence values of the tag are not significant.

Baked GI

Property	Function	
	Enlighten	**Progressive Lightmapper**
Blur Radius	The radius of the blur filter that is applied to direct lighting during post-processing in texels. The radius is essentially the distance over which neighboring texels are averaged out. A larger radius gives a more blurred effect. Higher levels of blur tend to reduce visual artifacts but also soften the edges of shadows.	**Blur Radius** is not available when you use Progressive **Lightmapper**.
Anti-aliasing Samples	The degree of anti-aliasing (the reduction of "blocky" artifacts) that is applied. Higher numbers increase quality and bake time.	The number of times to supersample a texel to reduce aliasing. Samples [1;3] disables supersampling, samples [4;8] give 2x supersampling, and samples [9;256] give 4x supersampling. This mostly affects the amount of memory used for the positions and normals buffers (2x uses four times the amount of memory, 4x uses 16 times the amount of memory).
Direct Light Quality	The number of rays used to evaluate direct lighting. A higher number of rays tends to produce more accurate soft shadows but increases bake time.	**Direct Light Quality** is not available when you use Progressive Lightmapper.

Property	Function	
Backface Tolerance	The structure of a Mesh sometimes causes some texels to have a “view” that includes back-facing geometry. Incoming light from a backface is meaningless in any Scene. Because of this, this property lets you select a percentage threshold of light that must come from front-facing geometry in order for a texel to be considered valid. Invalid texels have their lighting approximated from their neighbors’ values. Lowering this value can solve lighting problems caused by incoming light from backfaces.	The percentage of rays shot from an output texel that must hit front faces to be considered usable. Allows a texel to be invalidated if too many of the rays cast from it hit backfaces (the texel is inside some geometry). In that case artefacts are avoided by cloning valid values from surrounding texels. For example, if backface tolerance is 0.0, the texel is rejected only if it sees nothing but backfaces. If it is 1.0, the ray origin is rejected if it has even one ray that hits a backface. In the Baked Texel Validity **scene view** mode one case see valid (green) and invalid (red) texels. If you have a single sided mesh in your scene, you may want to disable this feature by setting it to zero. A two-sided flag will later be added in the editor to address this.
Baked Tag	Similar to the **System Tag** property above, this number lets you group specific sets of objects together in separate baked lightmaps. As with the System Tag, the exact numeric value is not significant. Objects with different Baked Tag values are never put in the same atlas; however, there is no guarantee that objects with the same tag end up in the same atlas, because those objects might not necessarily fit into one lightmap (see image A, below, for an example of this). You don’t have to set this when using the multi-scene bake API, because grouping is done automatically (use the **Baked Tag** to replicate some of the behavior of the **Lock Atlas** option). See Baked Tags: Details, below, for more information.	
Pushoff	The distance to push away from the surface geometry before starting to trace rays in modelling units. It is applied to all baked lightmaps, so it affects direct light, indirect light, and AO. **Pushoff** is useful for getting rid of unwanted AO or shadowing. Use this setting to solve problems where the surface of an object is shadowing itself, causing speckled shadow patterns to appear on the surface with no apparent source. You can also use this setting to remove unwanted artifacts on huge objects, where floating point precision isn’t high enough to accurately ray-trace fine detail.	The amount to push off ray origins away from geometry along the normal for ray tracing, in modelling units. It is applied to all baked lightmaps, so it affects direct light, indirect light and **ambient occlusion**. It is useful for getting rid of unwanted occlusion/shadowing.

Baked Tags: Details



The image above shows two views of the same Scene:

Left: Everything is in one atlas because all the GameObjects have the same **Baked Tag**.

Right: One GameObject is assigned a different **Baked Tag**, and forced into a second lightmap.

Baked AO

Property	Function
Quality	The number of rays cast when evaluating ambient occlusion (AO). A higher numbers of rays increases AO quality but also increases bake time.
Anti-aliasing Samples	The number of samples to take when doing anti-aliasing of AO. A higher number of samples increases the AO quality but also increases the bake time.

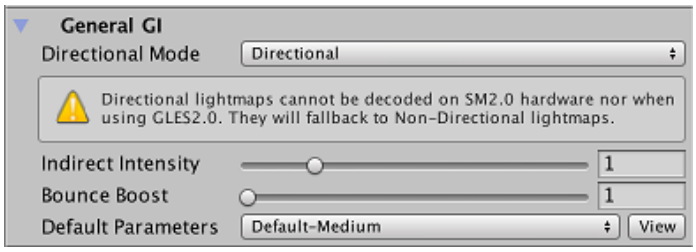
General GI

Property	Function
Backface Tolerance	The percentage of rays shot from an output texel that must hit front faces for the lighting system to consider them usable. This allows Unity to invalidate a texel if too many of the rays cast from it hit back faces ((for example, if the texel is inside some geometry). The lighting system clones valid values from the surrounding texels to avoid unintended artifacts. If **Backface Tolerance** is 0.0, the lighting system rejects the texel only if it sees nothing but backfaces. If it is 1.0, the lighting system rejects the ray origin if it has even one ray that hits a backface.

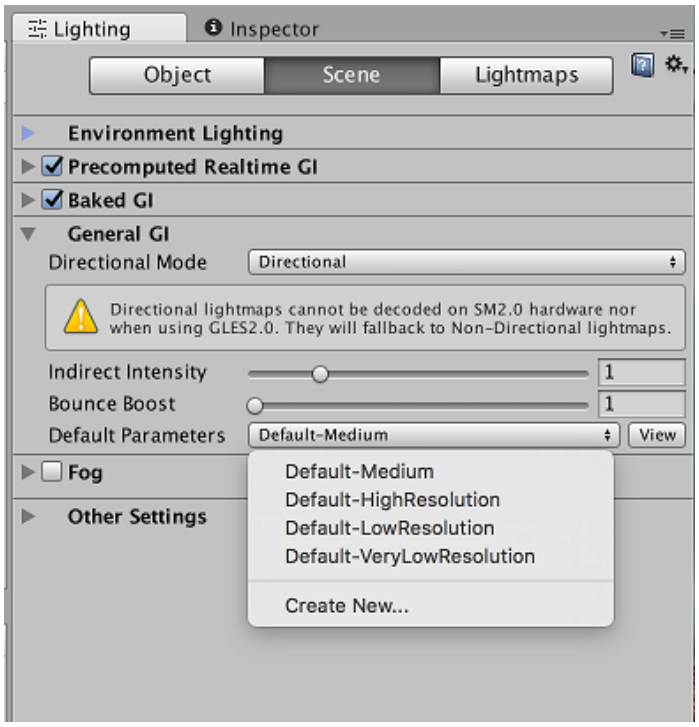
Assigning Lightmap Parameters Assets

Scenes

To assign a Lightmap Parameters Asset to the whole Scene, open the Lighting window (**Window** > **rendering** > **Lighting Settings**), click the **Scene** tab, and navigate to the **General GI** settings.

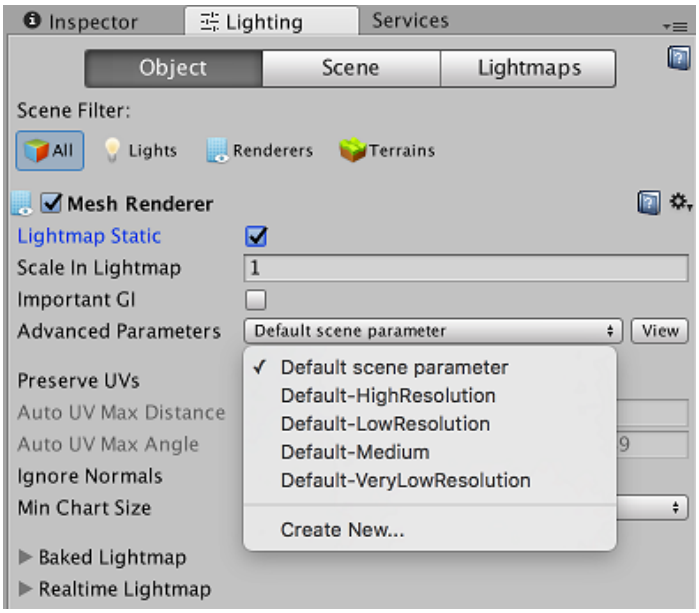


Use the **Default Parameters** drop-down to assign a default Lightmap Parameters Asset. This drop-down lists all available Lightmap Parameters Assets.

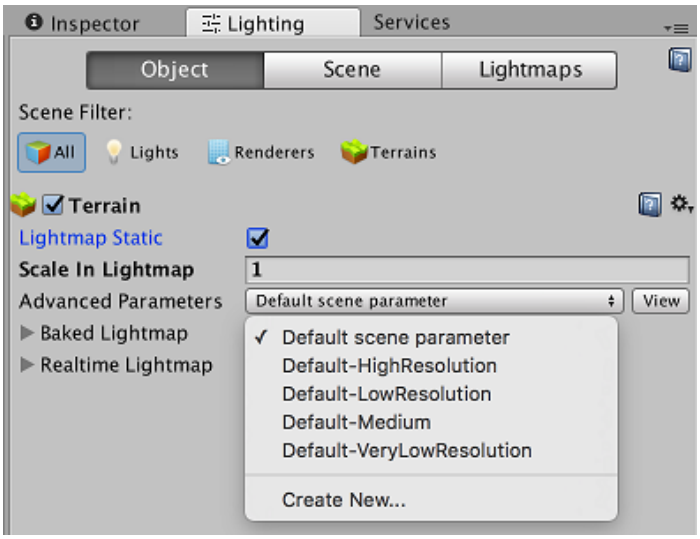


GameObjects

To assign a Lightmap Parameters Asset to a single GameObject, ensure the GameObject has a **Mesh Renderer** or **Terrain component** attached.



To assign a Lightmap Parameters Asset to a Mesh Renderer, tick the component's **Lightmap Static** checkbox and select an option from **Lightmap Parameters** under **Lightmap Settings**. Choose **Scene Default Parameter** to use the same Lightmap Parameters Asset that is assigned to the whole Scene.



To assign a Lightmap Parameters Asset to a Terrain, tick the Terrain's **Lightmap Static** checkbox and select an option from **Advanced Parameters**. Choose **Scene Default Parameters** to use the same Lightmap Parameters Asset that is assigned to the whole Scene.

2018–03–28 Page amended with limited editorial review

Progressive Lightmapper added in 2018.1

Baked ambient occlusion

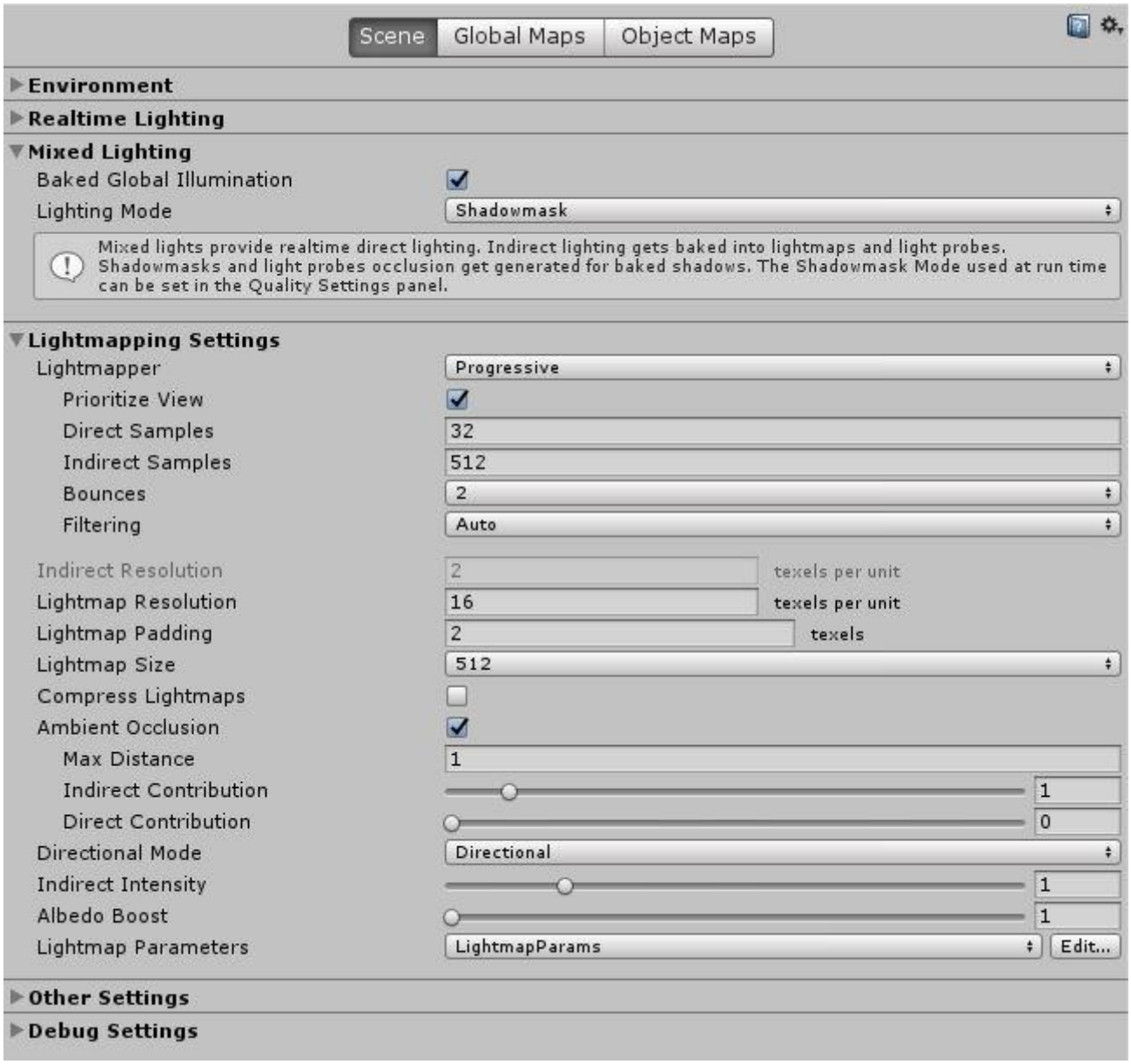
Leave feedback

Ambient occlusion (AO) approximates how much ambient lighting (lighting not coming from a specific direction) can hit a point on a surface. It darkens creases, holes and surfaces that are close to each other. These areas occlude (block out) ambient light, so they appear darker.

When using only precomputed real-time GI (see documentation on Using precomputed lighting), the resolution for indirect lighting doesn't capture fine details or dynamic objects. We recommend using a real-time ambient occlusion post effect, which has much more detail and results in higher quality lighting.

To view and enable baked AO, open the **Lighting** window (menu: **Window** > **Rendering** > **Lighting Settings**) and navigate to the **Baked GI** section. Tick the **Baked GI** checkbox if it is unchecked, then tick the **Ambient Occlusion** checkbox to enable baked AO.

To view and enable baked AO, open the **Lighting** window (menu: **Window** > **Rendering** > **Lighting Settings**) and navigate to the **Mixed Lighting** section. Tick the **Baked Global Illumination** checkbox if it is unchecked, then in **Lightmapping Settings** tick the **Ambient Occlusion** checkbox to enable baked AO.



Property	Function
Max Distance	Use this to set a value for how far the rays are traced before they are terminated.
Indirect	Use this to control how much the AO affects indirect light. The higher you set the slider value, the darker the appearance of the creases, holes, and close surfaces are when lit by indirect light. It's more realistic to only apply AO to indirect lighting.
Direct	Use this to control how much the AO affects direct light. The higher you set the slider value, the darker the appearance of the Scene's creases, holes, and close surfaces are when lit by direct light. By default, AO does not affect direct lighting. Use this slider to enable it. It is not realistic, but it can be useful for artistic purposes.

For a modern implementation of real-time ambient occlusion, see documentation on the Ambient Occlusion post-processing effect.

To learn more about AO, see Wikipedia: Ambient Occlusion.

LOD for baked lightmaps

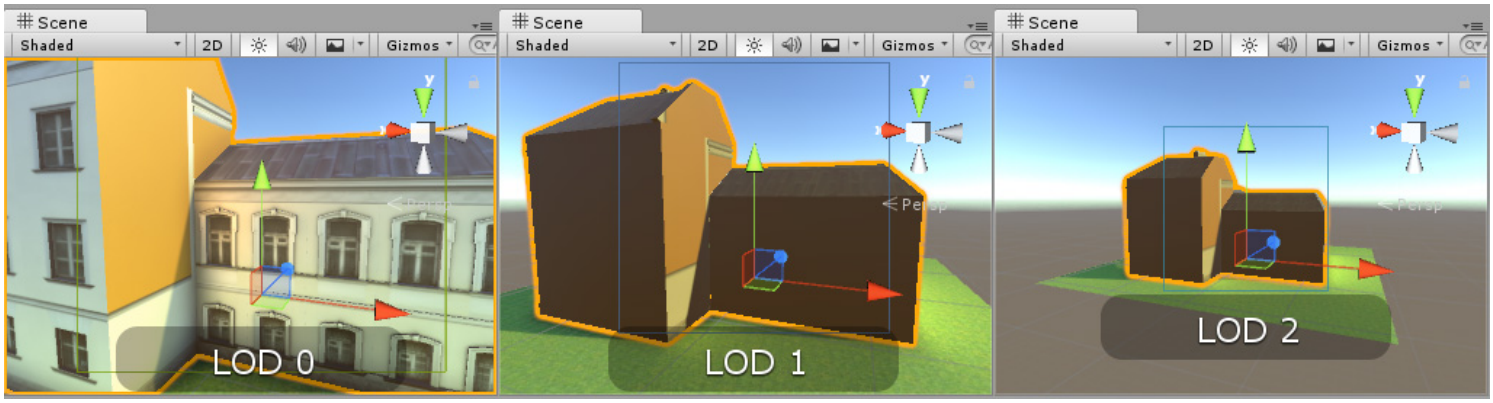
Leave feedback

This page provides advice on baking light into models that use Unity's LOD (level of detail) system.

When you use Unity's **LOD** system in a **Scene** with baked lighting, the system lights the most detailed model out of the **LOD Group** as if it is a regular static model. It uses **lightmaps** for the direct and indirect lighting, and separate **lightmaps** for Realtime GI.

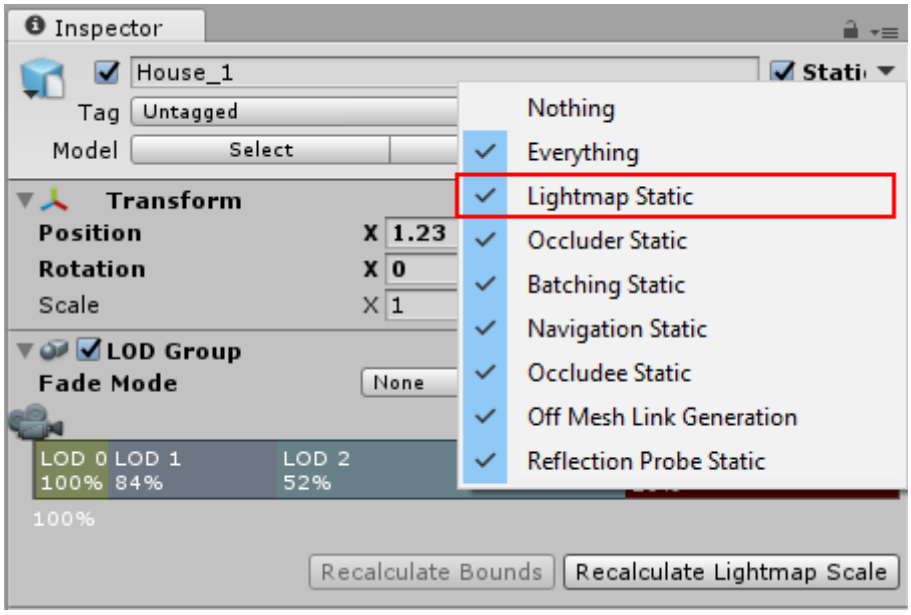
When you use the **Enlighten lightmapper**, the system only bakes the direct lighting, and the LOD system relies on Light Probes to sample indirect lighting.

To make sure your lower LOD models look correct with baked light, you must position Light Probes around them to capture the indirect lighting during the bake. Otherwise, your lower LOD models look incorrect, because they only receive direct light:



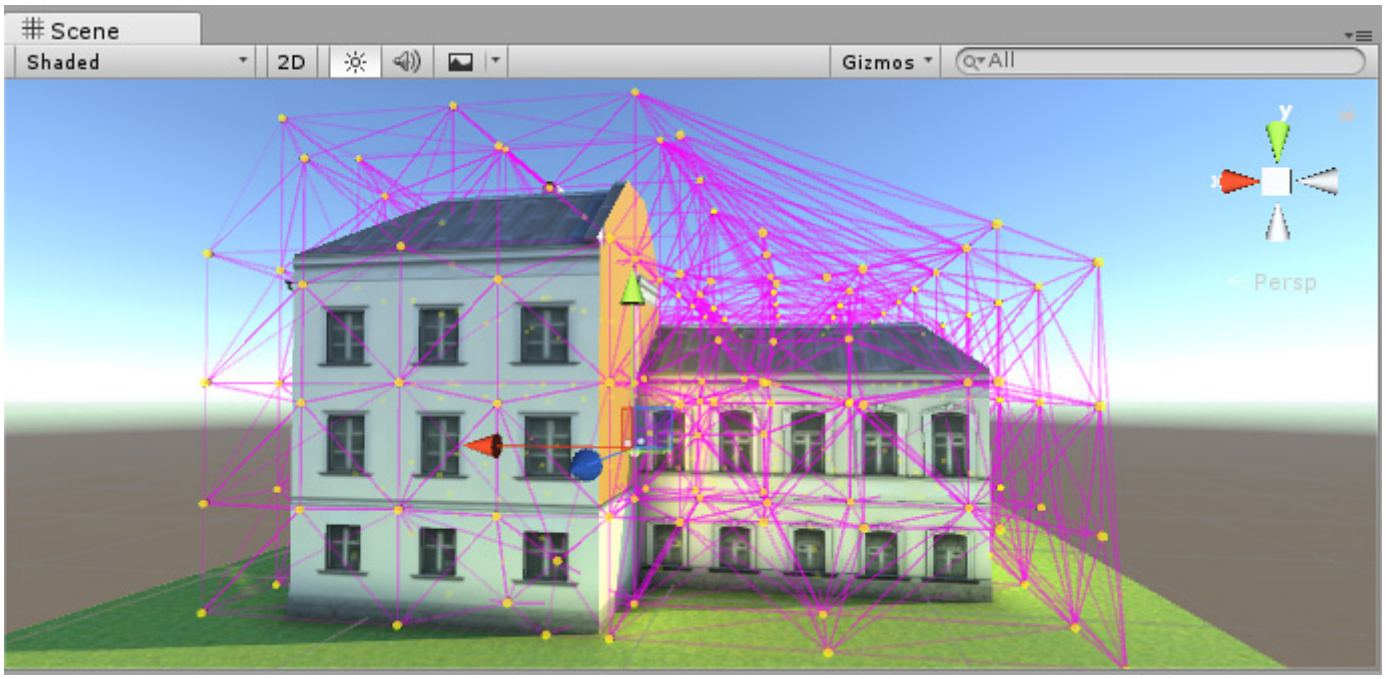
The LOD 1 and LOD 2 models here are lit incorrectly because light probes have not been placed around the model in the scene. They only show direct lighting.

To set up LOD models correctly for baked lighting, mark the LOD **GameObjects** as **Lightmap Static**. To do this, select the GameObject, and at the top of the **Inspector** window, select the drop-down menu next to the **Static** checkbox:



In this example, the LODs are assumed to be children of this GameObject

Use the Light Probes component to place Light Probes around the LOD GameObjects.



Light probes placed around an LOD model.

Note: Only the most detailed model affects the lighting on the surrounding geometry (for example, shadows or bounced light on surrounding buildings). In most cases this should not be a problem, because your lower level-of-detail models should closely resemble the highest level-of-detail model.

When you use the **Progressive Lightmapper**, there is no need to place Light Probes around the LOD Group to generate baked indirect lighting. However, to make Realtime GI affect the Renderers in the LOD Group, you must include the Light Probes.

2017–10–20 Page amended with editorial review

Updated in 5.6

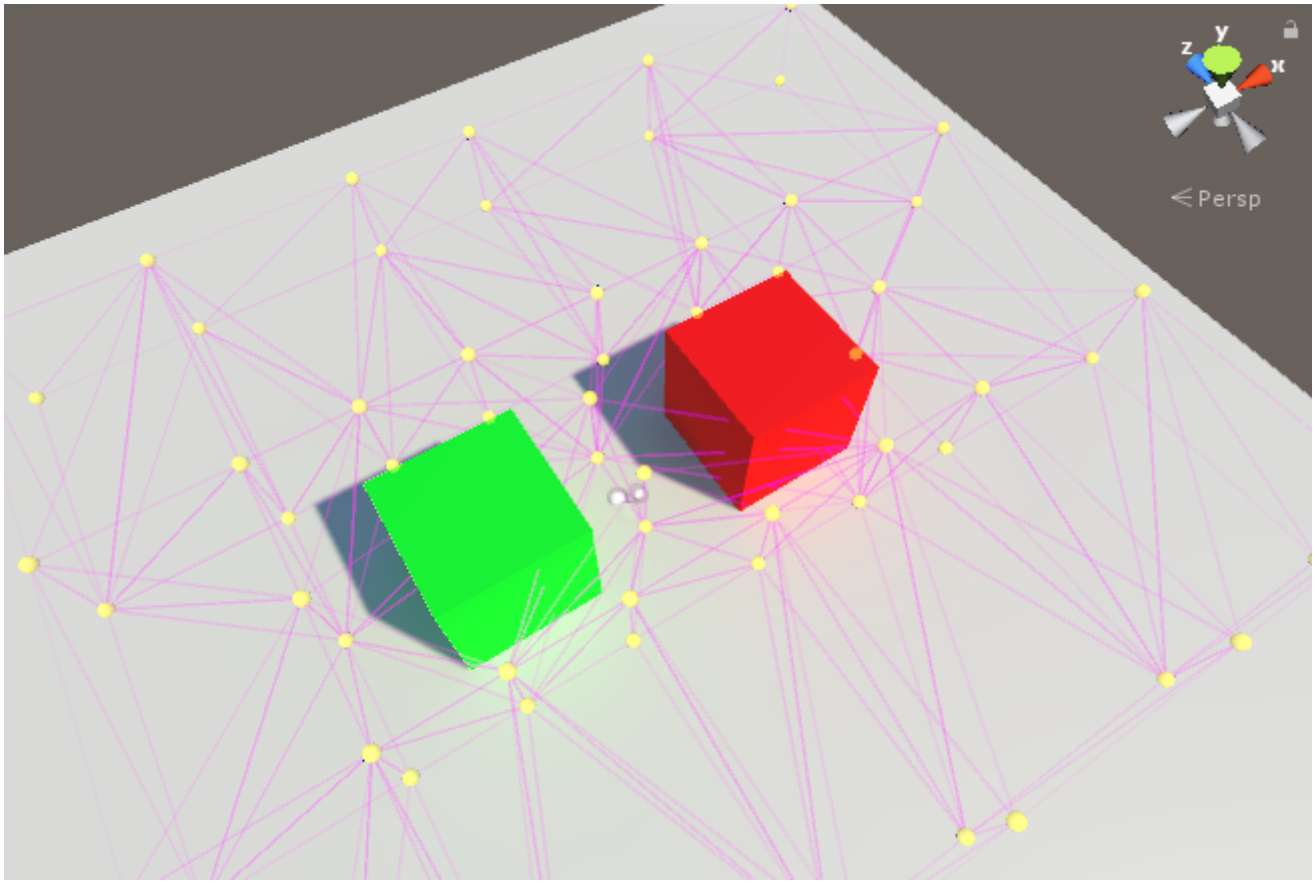
Updated in 2017.3

Light Probes

Leave feedback

Light Probes provide a way to capture and use information about light that is passing through the empty space in your **scene**.

Similar to **lightmaps**, light probes store "baked" information about lighting in your scene. The difference is that while **lightmaps** store lighting information about light hitting the *surfaces* in your scene, light probes store information about light passing through *empty space* in your scene.



An extremely simple scene showing light probes placed around two cubes

Light Probes have two main uses:

The primary use of light probes is to provide high quality lighting (including indirect bounced light) on moving objects in your scene.

The secondary use of light probes is to provide the lighting information for static scenery when that scenery is using Unity's **LOD system**.

When using light probes for either of these two distinct purposes, many of the techniques you need to use are the same. It's important to understand how light probes work so that you can choose where to place your probes in the scene.

2017–06–08 Page published with no editorial review

Light Probes updated in 5.6

Light Probes: Technical information

Leave feedback

The lighting information in the **light probes** are encoded as Spherical Harmonics basis functions. We use third order polynomials, also known as L2 Spherical Harmonics. These are stored using 27 floating point values, 9 for each color channel. Even though Unity is using Geomerics' **Enlighten**, we use a different SH basis than what you find on their blog (y and z axes are swapped). Unity is using the notation and reconstruction method from Peter-Pike Sloan's paper, Stupid Spherical Harmonics (SH) Tricks, and Geomerics are using the notation and reconstruction method from Ramamoorthi/Hanrahan's paper, An Efficient Representation for Irradiance Environment Maps.

The Unity **shader** code for reconstruction is found in UnityCG.cginc and is using the method from Appendix A10 **Shader**/CPU code for Irradiance Environment Maps from Peter-Pikes paper.

The data is internally ordered like this:

```
                        [L00:  DC]

            [L1-1:  y] [L10:   z] [L11:   x]

  [L2-2: xy] [L2-1: yz] [L20:  zz] [L21:  xz]  [L22:  xx - yy]
```

The 9 coefficients for R, G and B are ordered like this:

```
L00, L1-1,  L10,  L11, L2-2, L2-1,  L20,  L21,  L22, // red channel

L00, L1-1,  L10,  L11, L2-2, L2-1,  L20,  L21,  L22, // blue channel

L00, L1-1,  L10,  L11, L2-2, L2-1,  L20,  L21,  L22  // green channel
```

For more "under-the-hood" information about Unity's light probe system, you can read Robert Cupisz's talk from GDC 2012, Light Probe Interpolation Using Tetrahedral Tessellations", GDC 2012

2017–06–08 Page published with no editorial review

Light Probes updated in 5.6

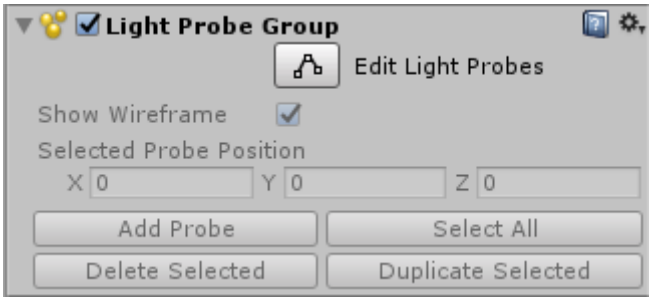
Placing Light Probes

Leave feedback

SWITCH TO SCRIPTING

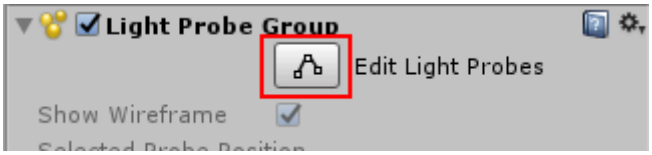
To place Light Probes in your scene, you must use a GameObject with a **Light Probe Group** component attached. You can add a Light Probe Group component from the menu: **Component -> Rendering -> Light Probe Group**.

The **Light Probe** Group **component** can be added to any object in the **scene**, however it's best to add it to a new empty **GameObject**.



The Light Probe Group component

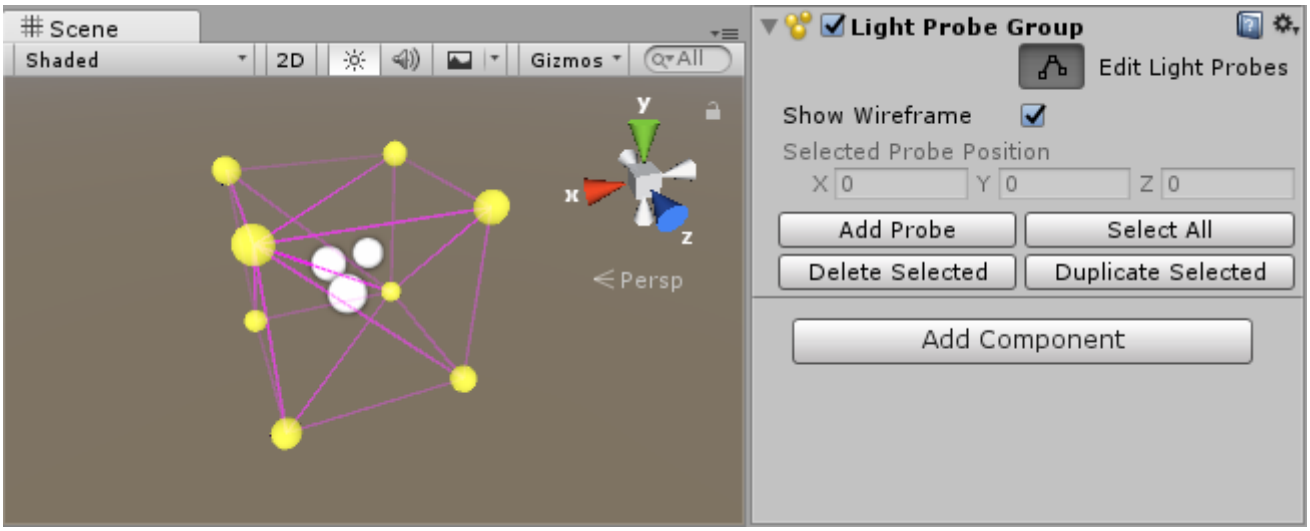
The Light Probe Group has its own **Edit Mode** which can be turned on or off. To add, move, or delete light probes, you must switch the Light Probe Group's edit mode on by pressing the **Edit Light Probes** button:



The Edit Light Probes button

When you are using the Edit Light Probes mode, you can manupulate individual light probes as in a similar way to GameObjects, however the individual probes are not GameObjects - they are stored as a set of points in the Light Probe Group component.

When you begin editing a new Light Probe Group, you will start off with a default formation of eight probes arranged in a cube, as shown below:



The default arrangement of light probes.

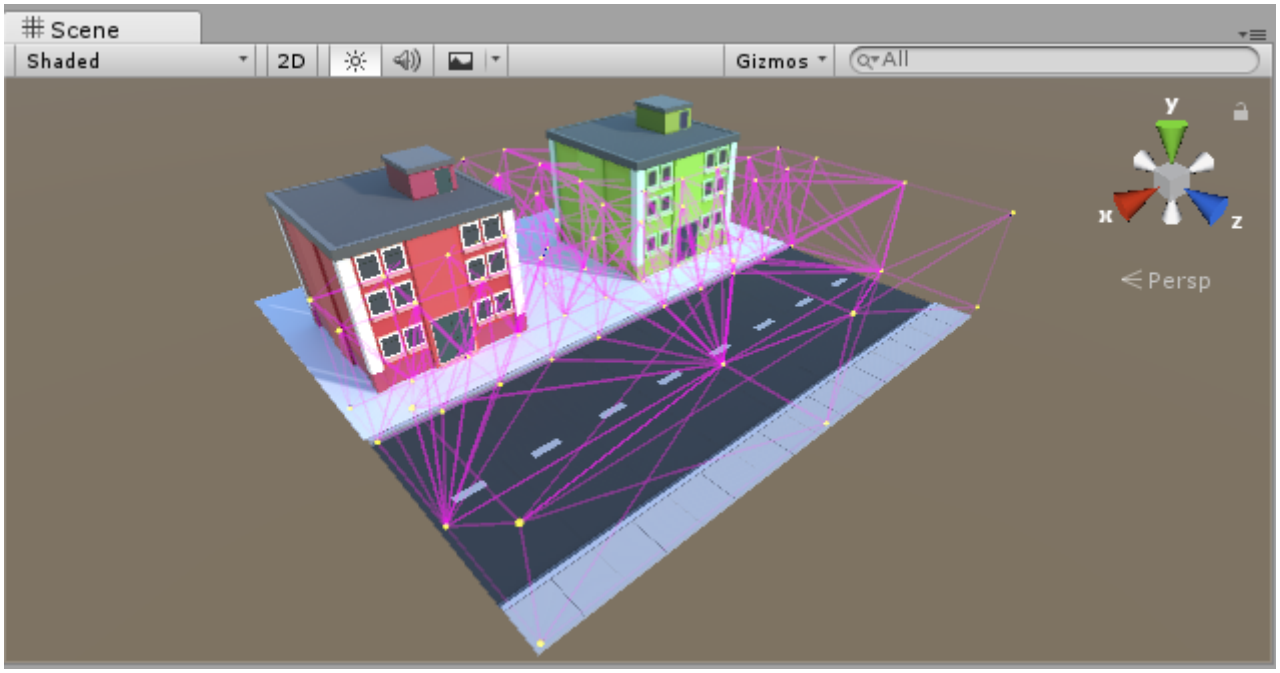
You can now use the controls in the Light Probe Group **inspector** to add new probe positions to the group. The probes appear in the scene as yellow spheres which can be positioned in the same manner as GameObjects. You can also select and duplicate individual probes or in groups, by using the usual “duplicate” keyboard shortcut (ctrl+d/cmd+d).

Remember to switch the Light Probe Group edit mode **off** once you’ve finished editing the probes, or you will find you cannot move or manipulate normal GameObjects!

Choosing Light Probe positions

Unlike **lightmaps**, which usually have a continuous resolution across the surface of an object, the resolution of the light probe information is entirely defined by how closely packed you choose to position the probes.

To optimise the amount of data that is stored by light probes, and the amount of computation done while the game is playing, you should generally attempt to place as few light probes as possible. However, you should also place enough probes that changes in light from one space to another are recorded at a level that is acceptable to you. This means you might place light probes in a more condensed pattern around areas that have complex or highly contrasting light, and you might place them in a much more spread out pattern over areas where the light does not significantly change.



Light probes placed with varying density around a simple scene

In the example above, the probes are placed more densely near and between the buildings where there is more contrast and color variation, and less densely along the road, where the lighting does not significantly change.

The simplest approach to positioning probes is to arrange them in a regular 3D grid pattern. While this setup is simple and effective, it is likely to consume more memory than necessary, and you may have lots of redunant probes. For example, in the scene above, if there were lots of probes placed along the road it would be a waste of resources. The light does not change much along the length of the road, so many probes would be storing almost identical lighting data to their neighbouring probes. In situations like this, it is much more efficient to interpolate this lighting data between fewer more spread-out probes.

Light probes individually do not store a large amount of information. From a technical perspective, each probe is a spherical, panoramic **HDR** image of the view from the sample point, encoded using Spherical Harmonics L2 which is stored as 27 floating point values. However in large scenes with hundreds of probes they can add up, and having unnecessarily densely packed probes can result in large amounts of wasted memory in your game.

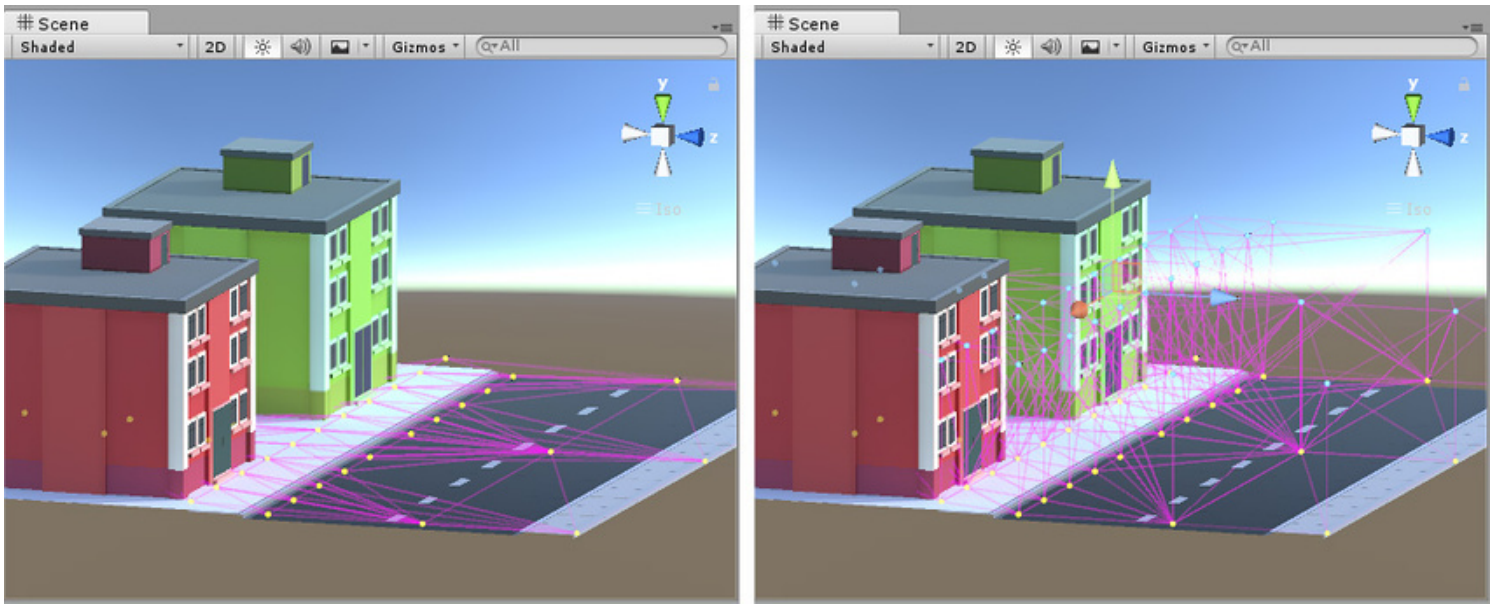
Creating a volume

Even if your gameplay takes place on a 2D plane (for example, cars driving around on a road surface), your light probes must form a 3D volume.

This means you should have at least two vertical “layers” of points in your group of probes.

In the example below, you can see on the left the probes are arranged only across the surface of the ground. This will **not result in good lighting** because the light probe system will not be able to calculat sensible tetrahedral volumes from the probes.

On the right, the probes are arranged in two layers, some low to the ground and others higher up, so that together they form a 3D volume made up of lots of individual tetrahedra. This is a **good** layout.



The left image shows a bad choice of light probe positions, because there is no height to the volume defined by the probes. The right image shows a good choice of light probe positions.

Light Probe placement for dynamic GI

Unity’s realtime GI allows moving lights to cast dynamic bounced light against your **static** scenery. However, you can also receive dynamic bounced light from moving lights on **moving GameObjects** when you are using llight probes.

Light Probes therefore perform two very similar but distinct functions - they store static baked light, and at runtime they represent sampling points for dynamic realtime global illumination (GI, or bounced light) to affect the lighting on moving objects.

Therefore, if you are using dynamic moving lights, and want realtime bounced light on your moving GameObjects, this may have implications on your choice of where you place your light probes, and how densely you group them.

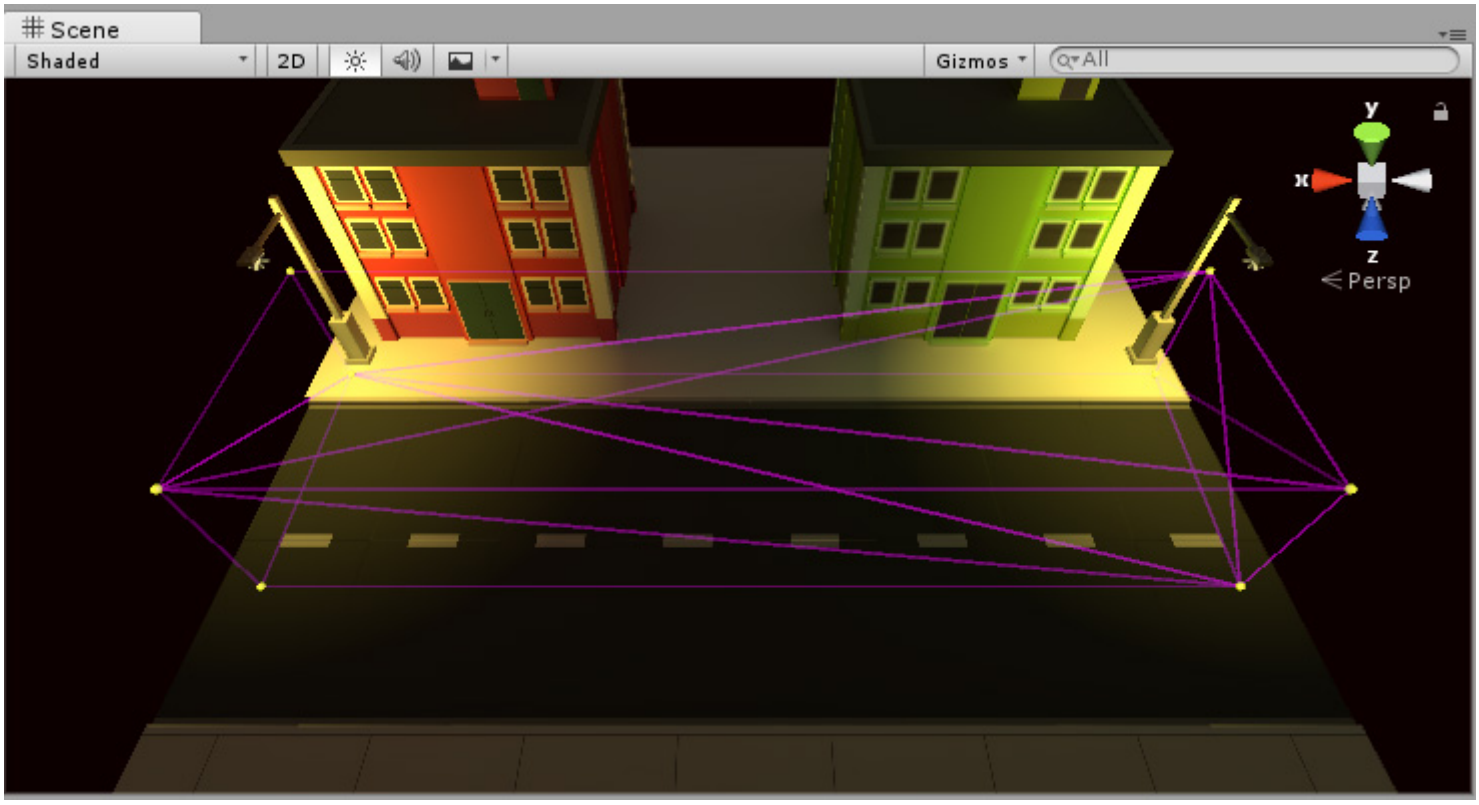
The main point to consider in this situation is that in large areas of relatively unchanging *static light* you might have placed only a few probes - because the light does not change across a wide area. However, if you plan to have *moving lights* within this area, and you want moving objects to receive bounced light from them, you will need a more dense network of light probes within the area so that there is a high enough level of accuracy to match your light's range and style.

How densely placed your probes need to be will vary depending on the size and range of your lights, how fast they move, and how large the moving objects are that you want to receive bounced light.

Light Probe placement problems

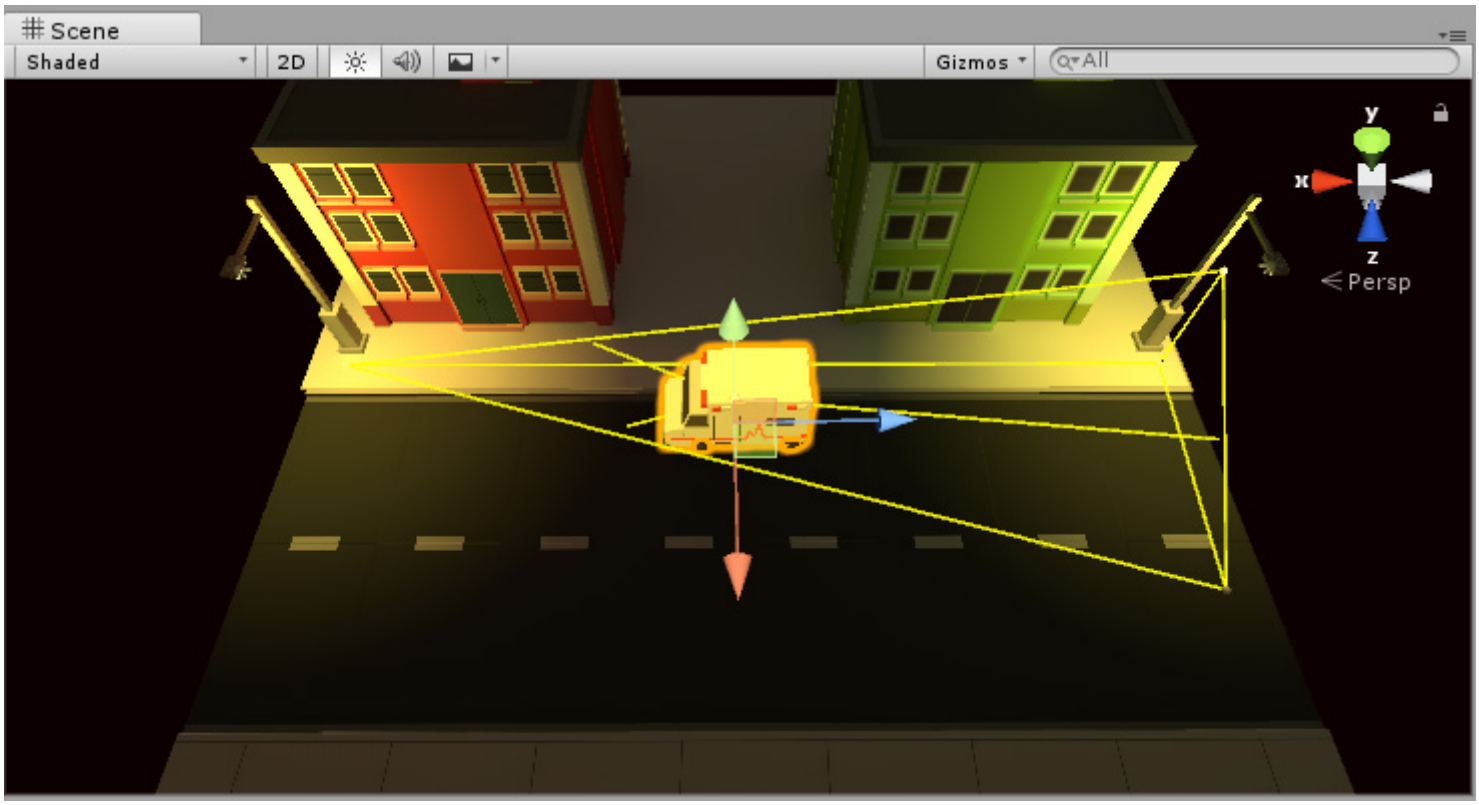
Your choice of light probe positions must take into account that the lighting will be interpolated between sets of probes. Problems can arise if your probes don't adequately cover the changes in lighting across your scene.

The example below shows a night-time scene with two bright street lamps on either side, and a dark area in the middle. If light probes are only placed near the street lamps, and none in the dark area, the lighting from the lamps will "bleed" across the dark gap, on moving objects. This is because the lighting is being interpolated from one bright point to another, with no information about the dark area in-between.



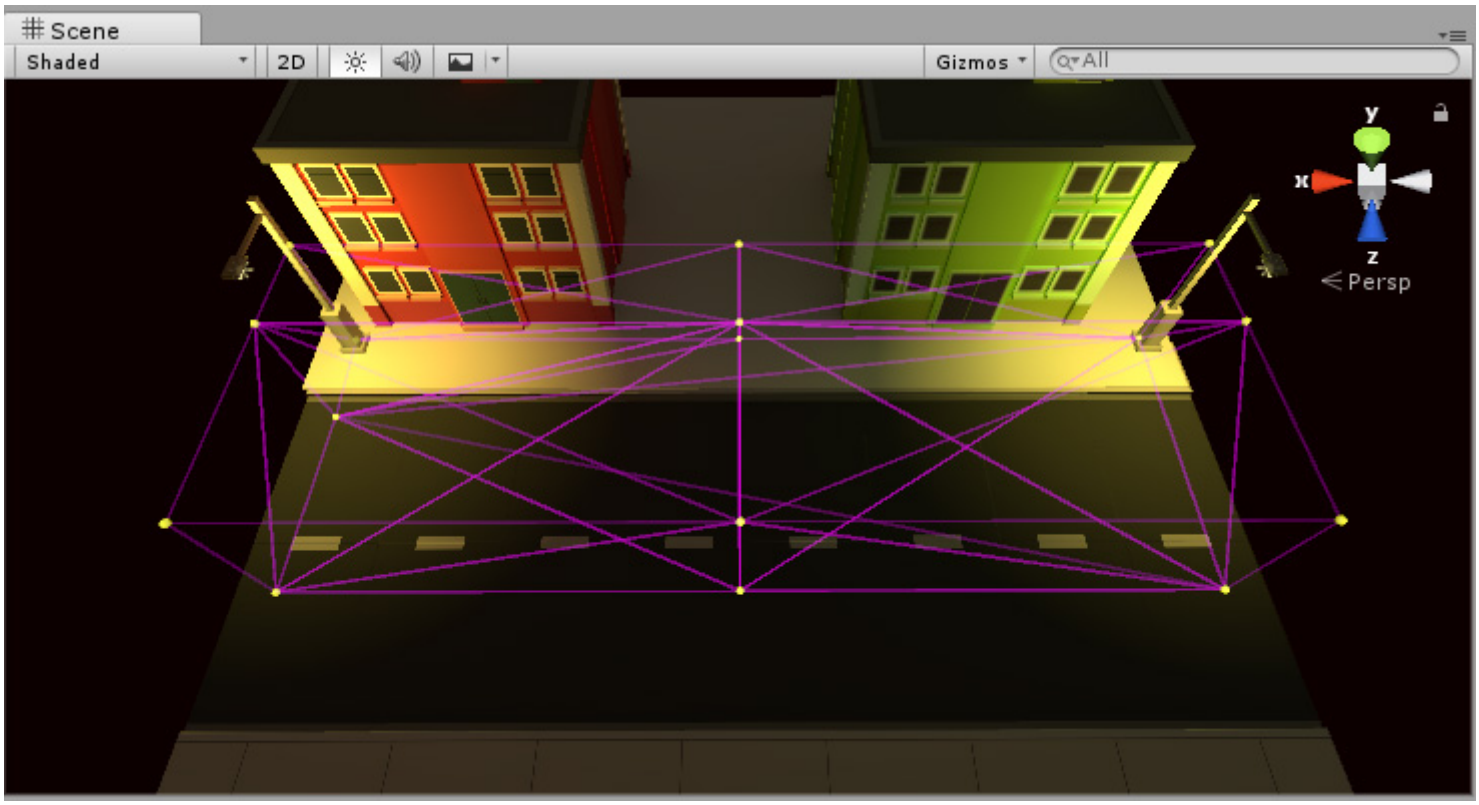
This image shows poor light probe placement. There are no probes in the dark area between the two lamps, so the dark area will not be included in the interpolation at all.

If you are using realtime or mixed lights, this problem may be less noticable, because only the *indirect* light will bleed across the gap. The problem is more noticable if you are using fully **baked lights**, because in this situation the direct light on moving objects is also interpolated from the light probes. In this example scene the two lamps are baked, so moving objects get their direct light from light probes. Here you can see the result - a moving object (the ambulance) remains brightly lit while passing through the dark area, which is not the desired effect. The yellow wireframe tetrahedron shows that the **interpolation** is occuring between one brightly lit end of the street to the other.

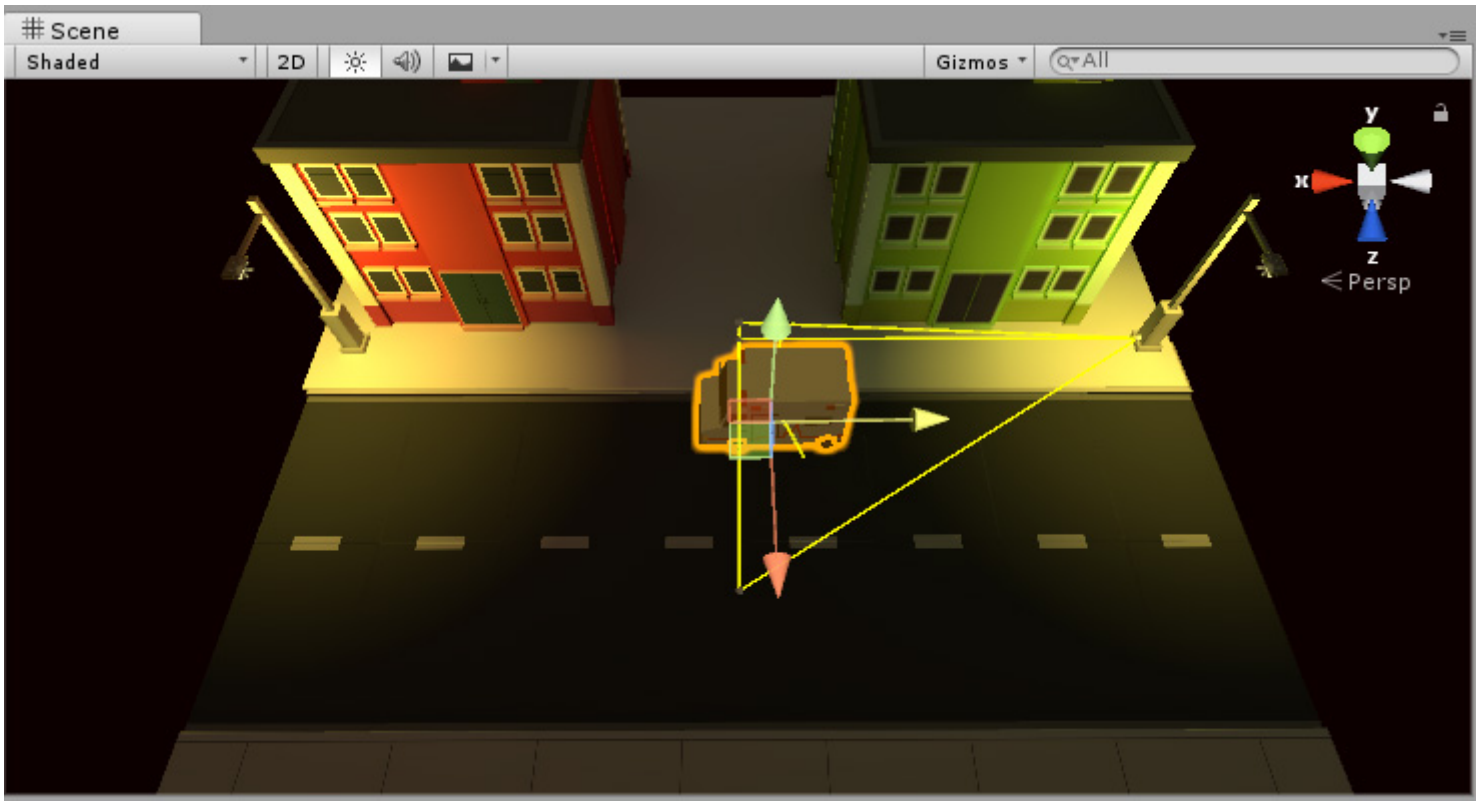


This is an undesired effect - the ambulance remains brightly lit while passing through a dark area, because no light probes were placed in the dark area.

To solve this, you should place more probes in the dark area, as shown below:



Now the scene has probes in the dark area too. And as a result, the moving ambulance takes on the darker lighting as it travels from one side of the scene to the other.



The ambulance now takes on the darker lighting in the centre of the scene, as desired.

2017–06–08 Page published with no editorial review

Light Probes updated in 5.6

Placing probes using scripting

Leave feedback

Placing **light probes** over large levels by hand can be time consuming. You can automate the placing of **light probes** by writing your own editor **scripts**. Your script can create a new **GameObject** with a LightProbeGroup **component**, and you can add probe positions individually according to any rules that you choose to program.

For example, this script can place Light Probes in a circle or a ring.

```
using UnityEngine;
using System.Collections.Generic;

[RequireComponent (typeof (LightProbeGroup))]
public class LightProbesTetrahedralGrid : MonoBehaviour
{
 // Common
 public float m_Side = 1.0f;
 public float m_Radius = 5.0f;
 public float m_InnerRadius = 0.1f;
 public float m_Height = 2.0f;
 public uint m_Levels = 3;
 const float kMinSide = 0.05f;
 const float kMinHeight = 0.05f;
 const float kMinInnerRadius = 0.1f;
 const uint kMinIterations = 4;
 public void OnValidate ()
 {
  m_Side = Mathf.Max (kMinSide, m_Side);
  m_Height = Mathf.Max (kMinHeight, m_Height);
  if (m_InnerRadius < kMinInnerRadius)
  {
   TriangleProps props = new TriangleProps (m_Side);
   m_Radius = Mathf.Max (props.circumscribedCircleRadius + 0.01f, m_Radius);
  }
  else
  {
   m_Radius = Mathf.Max (0.1f, m_Radius);
   m_InnerRadius = Mathf.Min (m_Radius, m_InnerRadius);
  }
 }
 struct TriangleProps
 {
  public TriangleProps (float triangleSide)
  {
   side = triangleSide;
   halfSide = side / 2.0f;
   height = Mathf.Sqrt (3.0f) * side / 2.0f;
```

```
  inscribedCircleRadius = Mathf.Sqrt (3.0f) * side / 6.0f;
  circumscribedCircleRadius = 2.0f * height / 3.0f;
 }
 public float side;
 public float halfSide;
 public float height;
 public float inscribedCircleRadius;
 public float circumscribedCircleRadius;
};

private TriangleProps m_TriangleProps;
public void Generate ()
{
 LightProbeGroup lightProbeGroup = GetComponent<LightProbeGroup> ();
 List<Vector3> positions = new List<Vector3> ();
 m_TriangleProps = new TriangleProps (m_Side);
 if (m_InnerRadius < kMinInnerRadius)
  GenerateCylinder (m_TriangleProps, m_Radius, m_Height, m_Levels, positions);
 else
  GenerateRing (m_TriangleProps, m_Radius, m_InnerRadius, m_Height, m_Levels, p
 lightProbeGroup.probePositions = positions.ToArray ();
}
static void AttemptAdding (Vector3 position, Vector3 center, float distanceCuto
{
 if ((position - center).sqrMagnitude < distanceCutoffSquared)
  outPositions.Add (position);
}
uint CalculateCylinderIterations (TriangleProps props, float radius)
{
 int iterations = Mathf.CeilToInt ((radius + props.height - props.inscribedCirc
 if (iterations > 0)
  return (uint)iterations;
 return 0;
}
void GenerateCylinder (TriangleProps props, float radius, float height, uint le
{
 uint iterations = CalculateCylinderIterations (props, radius);
 float distanceCutoff = radius;
 float distanceCutoffSquared = distanceCutoff * distanceCutoff;
 Vector3 up = new Vector3 (props.circumscribedCircleRadius, 0.0f, 0.0f);
 Vector3 leftDown = new Vector3 (-props.inscribedCircleRadius, 0.0f, -props.hal
 Vector3 rightDown = new Vector3 (-props.inscribedCircleRadius, 0.0f, props.hal
 for (uint l = 0; l < levels; l++)
 {
  float tLevel = levels == 1 ? 0 : (float)l / (float)(levels - 1);
  Vector3 center = new Vector3 (0.0f, tLevel * height, 0.0f);
```

```
if (l % 2 == 0)
{
 for (uint i = 0; i < iterations; i++)
 {
  Vector3 upCorner = center + up + (float)i * up * 2.0f * 3.0f / 2.0f;
  Vector3 leftDownCorner = center + leftDown + (float)i * leftDown * 2.0f * 3
  Vector3 rightDownCorner = center + rightDown + (float)i * rightDown * 2.0f
  AttemptAdding (upCorner, center, distanceCutoffSquared, outPositions);
  AttemptAdding (leftDownCorner, center, distanceCutoffSquared, outPositions)
  AttemptAdding (rightDownCorner, center, distanceCutoffSquared, outPositions
  Vector3 leftDownUp = upCorner - leftDownCorner;
  Vector3 upRightDown = rightDownCorner - upCorner;
  Vector3 rightDownLeftDown = leftDownCorner - rightDownCorner;
  uint subdiv = 3 * i + 1;
  for (uint s = 1; s < subdiv; s++)
  {
   Vector3 leftDownUpSubdiv = leftDownCorner + leftDownUp * (float)s / (float
   AttemptAdding (leftDownUpSubdiv, center, distanceCutoffSquared, outPositio
   Vector3 upRightDownSubdiv = upCorner + upRightDown * (float)s / (float)sub
   AttemptAdding (upRightDownSubdiv, center, distanceCutoffSquared, outPositi
   Vector3 rightDownLeftDownSubdiv = rightDownCorner + rightDownLeftDown * (f
   AttemptAdding (rightDownLeftDownSubdiv, center, distanceCutoffSquared, out
  }
 }
}
else
{
 for (uint i = 0; i < iterations; i++)
 {
  Vector3 upCorner = center + (float)i * (2.0f * up * 3.0f / 2.0f);
  Vector3 leftDownCorner = center + (float)i * (2.0f * leftDown * 3.0f / 2.0f
  Vector3 rightDownCorner = center + (float)i * (2.0f * rightDown * 3.0f / 2.
  AttemptAdding (upCorner, center, distanceCutoffSquared, outPositions);
  AttemptAdding (leftDownCorner, center, distanceCutoffSquared, outPositions)
  AttemptAdding (rightDownCorner, center, distanceCutoffSquared, outPositions
  Vector3 leftDownUp = upCorner - leftDownCorner;
  Vector3 upRightDown = rightDownCorner - upCorner;
  Vector3 rightDownLeftDown = leftDownCorner - rightDownCorner;
  uint subdiv = 3 * i;
  for (uint s = 1; s < subdiv; s++)
  {
   Vector3 leftDownUpSubdiv = leftDownCorner + leftDownUp * (float)s / (float
   AttemptAdding (leftDownUpSubdiv, center, distanceCutoffSquared, outPositio
   Vector3 upRightDownSubdiv = upCorner + upRightDown * (float)s / (float)sub
   AttemptAdding (upRightDownSubdiv, center, distanceCutoffSquared, outPositi
   Vector3 rightDownLeftDownSubdiv = rightDownCorner + rightDownLeftDown * (f
```

```
        AttemptAdding (rightDownLeftDownSubdiv, center, distanceCutoffSquared, out
       }
      }
     }
    }
   }
   void GenerateRing (TriangleProps props, float radius, float innerRadius, float
   {
    float chordLength = props.side;
    float angle = Mathf.Clamp (2.0f * Mathf.Asin (chordLength / (2.0f * radius)),
    uint slicesAtRadius = (uint)Mathf.FloorToInt (2.0f * Mathf.PI / angle);
    uint layers = (uint)Mathf.Max (Mathf.Ceil ((radius - innerRadius) / props.heig
    for (uint level = 0; level < levels; level++)
    {
     float tLevel = levels == 1 ? 0 : (float)level / (float)(levels - 1);
     float y = height * tLevel;
     float iterationOffset0 = level % 2 == 0 ? 0.0f : 0.5f;
     for (uint layer = 0; layer < layers; layer++)
     {
      float tLayer = layers == 1 ? 1.0f : (float)layer / (float)(layers - 1);
      float tIterations = (tLayer * (radius - innerRadius) + innerRadius - kMinInn
      uint slices = (uint)Mathf.CeilToInt (Mathf.Lerp (kMinIterations, slicesAtRad
      float x = innerRadius + (radius - innerRadius) * tLayer;
      Vector3 position = new Vector3 (x, y, 0.0f);
      float layerSliceOffset = layer % 2 == 0 ? 0.0f : 0.5f;
      for (uint slice = 0; slice < slices; slice++)
      {
       Quaternion rotation = Quaternion.Euler (0.0f, (slice + iterationOffset0 + l
       outPositions.Add (rotation * position);
      }
     }
    }
   }
  }
```

2017–06–08 Page published with no editorial review

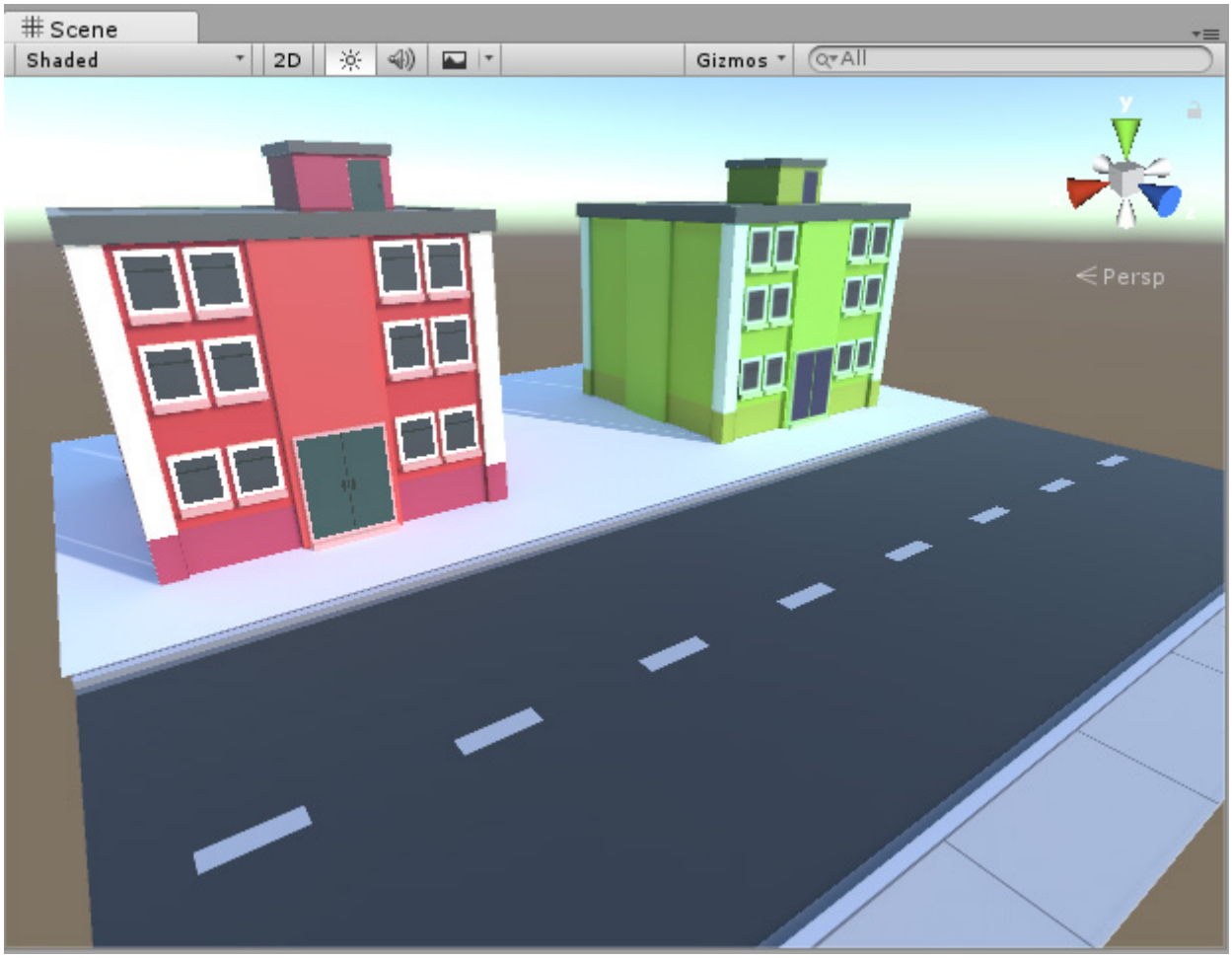
Light Probes updated in 5.6

Light Probes for moving objects

Leave feedback

Lightmapping adds greatly to the realism of a scene by capturing realistic bounced light as textures which are "baked" onto the surface of **static** objects. However, due to the nature of lightmapping, it can only be applied to non-moving objects marked as Lightmap Static.

While realtime and mixed mode lights can cast *direct* light on moving objects, moving objects do not receive bounced light from your static environment unless you use **light probes**. Light probes store information about how light is bouncing around in your **scene**. Therefore as objects move through the spaces in your game environment, they can use the information stored in your light probes to show an approximation of the bounced light at their current position.



A simple scene showing bounced light from static scenery.

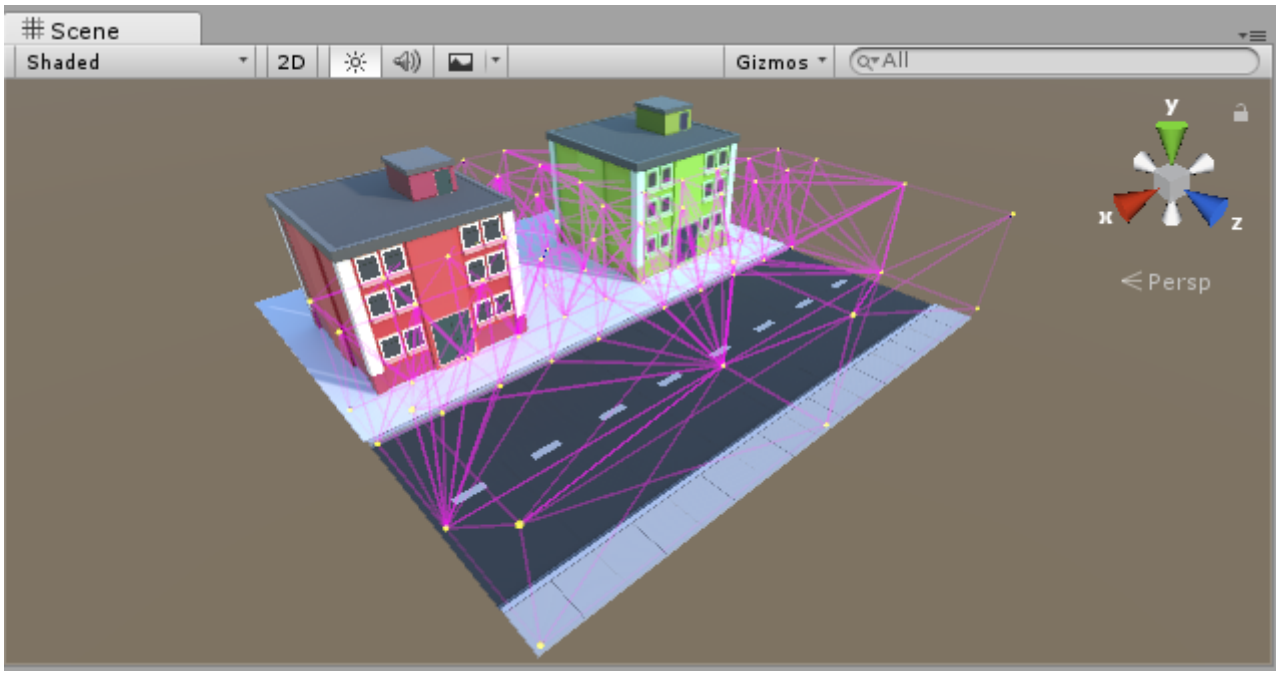
In the above scene, as the directional light hits the red and green buildings, which are static scenery, *bounced light* is cast into the scene. The bounced light is visible as a red and green tint on the ground directly in front of each building. Because all these models are **static**, all this lighting is stored in **lightmaps**.

When you introduce moving objects into your scene, they do not automatically receive bounced light. In the below image, you can see the ambulance (a dynamic moving object) is not affected by the bounced red light coming off the building. Instead, its side is a flat grey color. This is because the ambulance is a dynamic object which can move around in the game, and therefore cannot use lightmaps, because they are static by nature. The scene needs Light Probes so that the moving ambulance can receive bounced light.

The side of the ambulance is a flat grey color, even though it should be receiving some bounced red light from the front of the building.

To use the light probe feature to cast bounced light onto dynamic moving objects, you must position light probes throughout your scene, so that they cover the areas of space that moving objects in your game might pass through.

The probes you place in your scene define a 3D volume. The lighting at any position within this volume is then approximated on moving objects by interpolating between the information baked into the nearest probes.



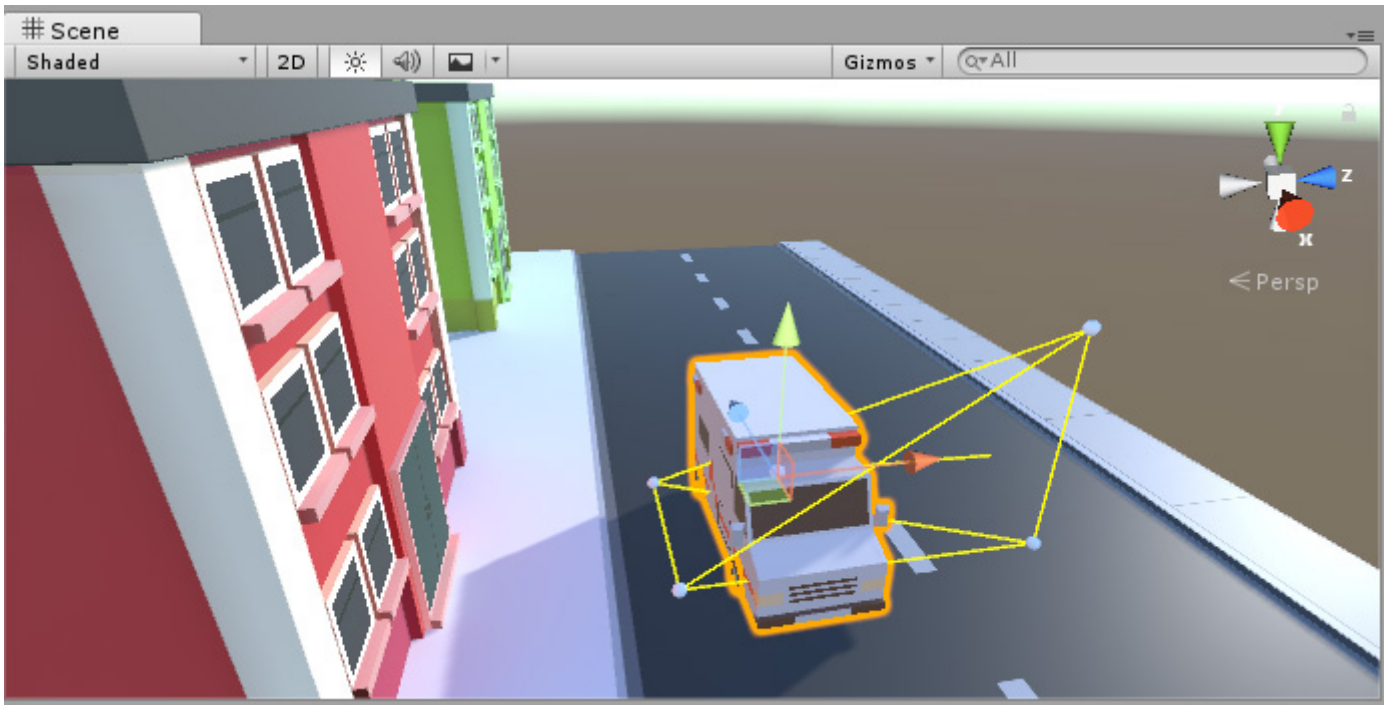
Light probes placed around the static scenery in a simple scene. The light probes are shown as yellow dots. They are shown connected by magenta lines, to visualise the volume that they define.

Once you have added probes, and baked the light in your scene, your dynamic moving objects will receive bounced light based on the nearest probes in the scene. Using the same example as above, the dynamic object

(the ambulance) now receives bounced light from the static scenery, giving the side of the vehicle a red tint, because it is in front of the red building which is casting bounced light.

The side of the ambulance now has a red tint because it is receiving bounced red light from the front of the building, via the light probes in the scene.

When a dynamic object is selected, the **Scene view** will draw a visualisation of which light probes are being used for the interpolated bounced light. The nearest probes to the dynamic object are used to form a tetrahedral volume, and the dynamic object's light is interpolated from the four points of this tetrahedron.



The light probes that are being used to light a dynamic object are revealed in the scene view when the object is selected, connected by yellow lines to show the tetrahedral volume.

As an object passes through the scene, it moves from one tetrahedral volume to another, and the lighting is calculated based on its position within the current tetrahedron.

A dynamic object moving through a scene with light probes, showing how it passes from one tetrahedral light probe volume to another.

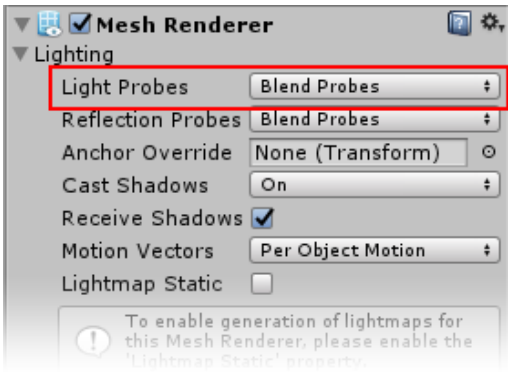
2017–06–08 Page published with no editorial review

Light Probes updated in 5.6

Light Probes and the Mesh Renderer

Leave feedback

To use **Light Probes** on your moving **GameObjects**, the **Mesh Renderer component** on the moving GameObject must set correctly. The **Mesh** Renderer **component** has a ***Light Probes*** setting which is set to **Blend Probes** by default. This means that by default, all **GameObjects** will use **light probes** and will blend between the nearest probes as it changes position in your **scene**.

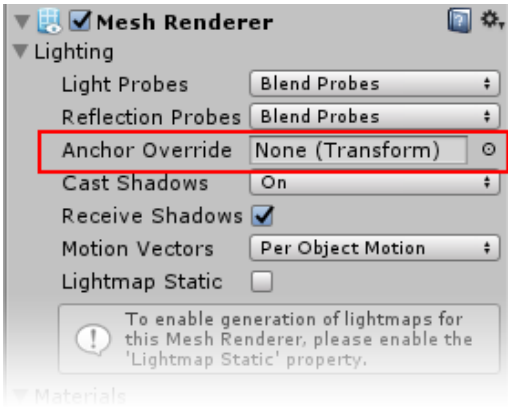


The Light Probes setting on the Mesh Renderer component.

You can change this setting to either “off” or “use proxy volume”. Switching the light probes setting to off will disable the light probe’s effect on this GameObject.

Light Probe Proxy Volumes are a special setting which you can use for situations where a **large moving object** might be too big to be sensibly lit by the results of a single tetrahedron from the **light probe group**, and instead needs to be lit by multiple groups of light probes across the length of the model. See **Light Probe Proxy Volumes** for more information.

The other setting in the Mesh Renderer **inspector** which relates to light probes is the **Anchor Override** setting. As described previously, when a GameObject moves through your scene, Unity calculates which tetrahedron the GameObject falls within from the volume defined by the light probe groups. By default this is calculated from the centre point of the Mesh’s bounding box, however you can override the point that is used by assigning a different GameObject to the **Anchor Override** field.



The Anchor Override setting in the Mesh Renderer component.

If you assign a different GameObject to this field, it is up to you to move that GameObject in a way that suits the lighting you want on your mesh.

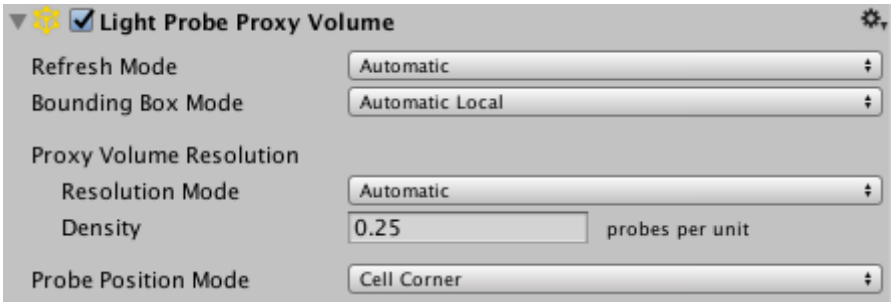
The anchor override may be useful when a GameObject contains two separate adjoining meshes; if both meshes are lit individually according to their bounding box positions then the lighting will be discontinuous at the place where they join. This can be prevented by using the same Transform (for example the parent or a child object) as the **interpolation** point for both Mesh Renderers or by using a Light Probe Proxy Volume.

2017–06–08 Page published with no editorial review

Light Probe Proxy Volume component

Leave feedback

SWITCH TO SCRIPTING



The **Light Probe Proxy Volume** (LPPV) **component** allows you to use more lighting information for large dynamic **GameObjects** that cannot use baked **lightmaps** (for example, large Particle Systems or skinned Meshes).

By default, a probe-lit Renderer receives lighting from a single Light Probe that is interpolated between the surrounding Light Probes in the **Scene**. Because of this, GameObjects have constant ambient lighting across the surface. This lighting has a rotational gradient because it is using spherical harmonics, but it lacks a spatial gradient. This is more noticeable on larger GameObjects or **Particle Systems**. The lighting across the GameObject matches the lighting at the anchor point, and if the GameObject straddles a lighting gradient, parts of the GameObject may look incorrect.

The **Light Probe Proxy Volume** component generates a 3D grid of interpolated Light Probes inside a **Bounding Volume**. You can specify the resolution of this grid in the **UI** of the component. The spherical harmonics (SH) coefficients of the interpolated Light Probes are uploaded into 3D textures. The 3D textures containing SH coefficients are then sampled at render time to compute the contribution to the diffuse ambient lighting. This adds a spatial gradient to probe-lit GameObjects.

The Standard Shader supports this feature. To add this to a custom **shader**, use the `ShadeSHPerPixel` function. To learn how to implement this function, see the Particle System sample Shader code example at the bottom of this page.

When to use the component

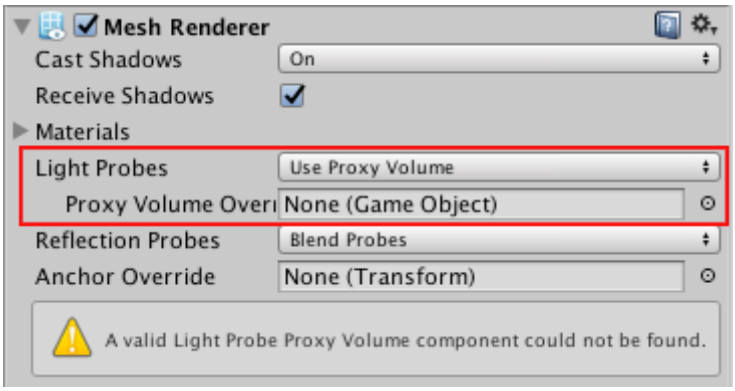
Most of the Renderer components in Unity contain Light Probes. There are three options for Light Probes:

Off: the Renderer doesn't use any interpolated Light Probes.

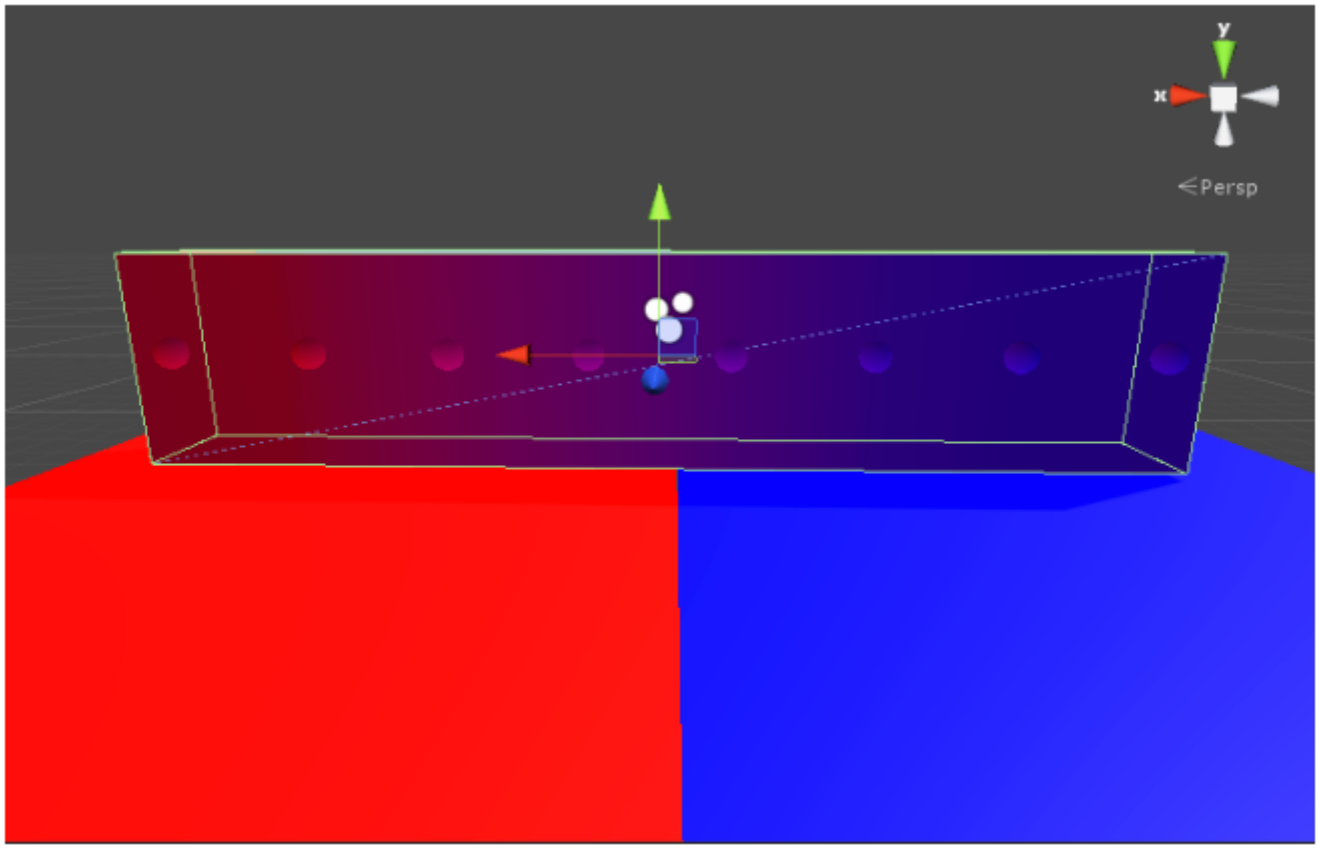
Blend Probes (default value): the Renderer uses one interpolated Light Probe.

Use Proxy Volume: the Renderer uses a 3D grid of interpolated Light Probes.

When you set the **Light Probes** property to **Use Proxy Volume**, the GameObject must have a **Light Probe Proxy Volume** (LPPV) component attached. You can add a LPPV component on the same GameObject, or you can use (borrow) a LPPV component from another GameObject using the **Proxy Volume Override** property. If Unity cannot find a LPPV component in the current GameObject or in the Proxy Volume Override GameObject, a warning message is displayed at the bottom of the Renderer.



Example



An example of a simple Mesh Renderer using a Light Probe Proxy Volume component

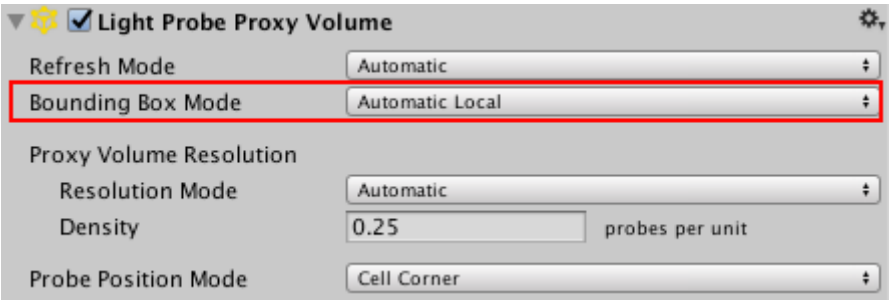
In the Scene above, there are two planes on the floor using Materials that emit a lot of light. Note that:

The ambient light changes across the geometry when using a LPPV component. Use one interpolated Light Probe to create a constant color on each side of the geometry.

The geometry doesn't use static lightmaps, and the spheres represent interpolated Light Probes. They are part of the **Gizmo Renderer**.

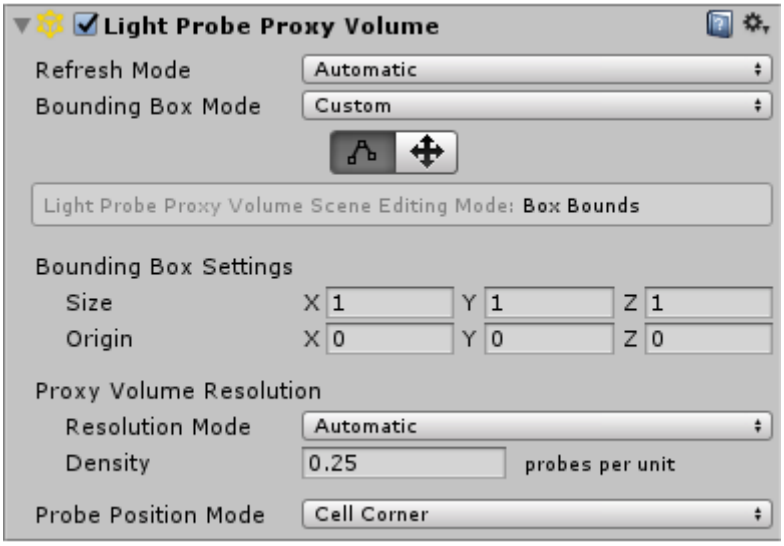
How to use the component

The area in which the 3D grid of interpolated Light Probes are generated is affected by the **Bounding Box Mode** property.



There are three options available:

Bounding Box Mode:	Function:
Automatic Local (default value)	A local-space bounding box is computed. The interpolated Light Probe positions are generated inside this bounding box. If a Renderer component isn't attached to the GameObject, then a default bounding box is generated. The bounding box computation encloses the current Renderer, and sets all the Renderers down the hierarchy that have the **Light Probes** property to **Use Proxy Volume**.
Automatic World	A bounding box is computed which encloses the current Renderer and all the Renderers down the hierarchy that have the **Light Probes** property set to **Use Proxy Volume**. The bounding box is world-aligned.
Custom	A custom bounding box is used. The bounding box is specified in the local-space of the GameObject. The bounding box editing tools are available. You can edit the bounding volume manually by modifying the **Size** and **Origin** values in the UI.

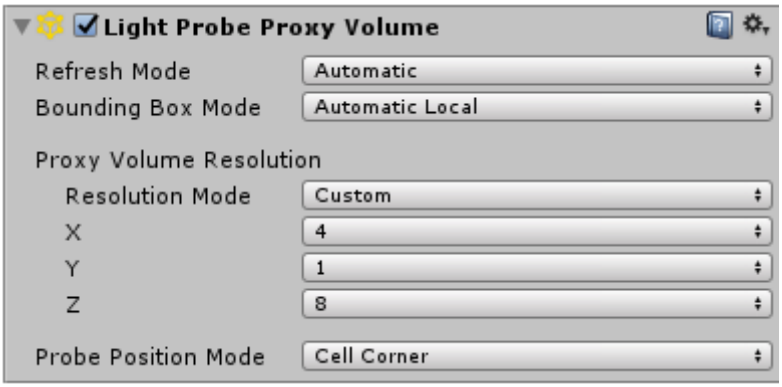


The main difference between **Automatic Local** and **Automatic World** is that in **Automatic Local**, the bounding box is more resource-intensive to compute when a large hierarchy of GameObjects uses the same LPPV component from a parent GameObject. However, the resulting bounding box may be smaller in size, meaning the lighting data is more compact.

The number of interpolated Light Probes from within the bounding volume is affected by the **Proxy Volume Resolution** property. There are two options:

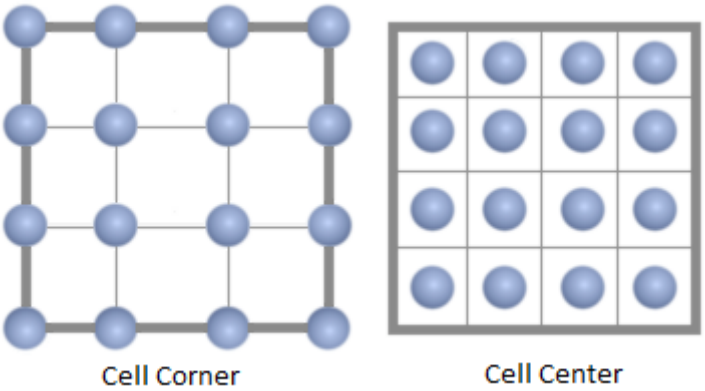
Automatic (default value) - The resolution on each axis is computed using the number of interpolated Light Probes per unit area that you specify, and the size of the bounding box.

Custom - Allows you to specify a different resolution on each axis (see below).



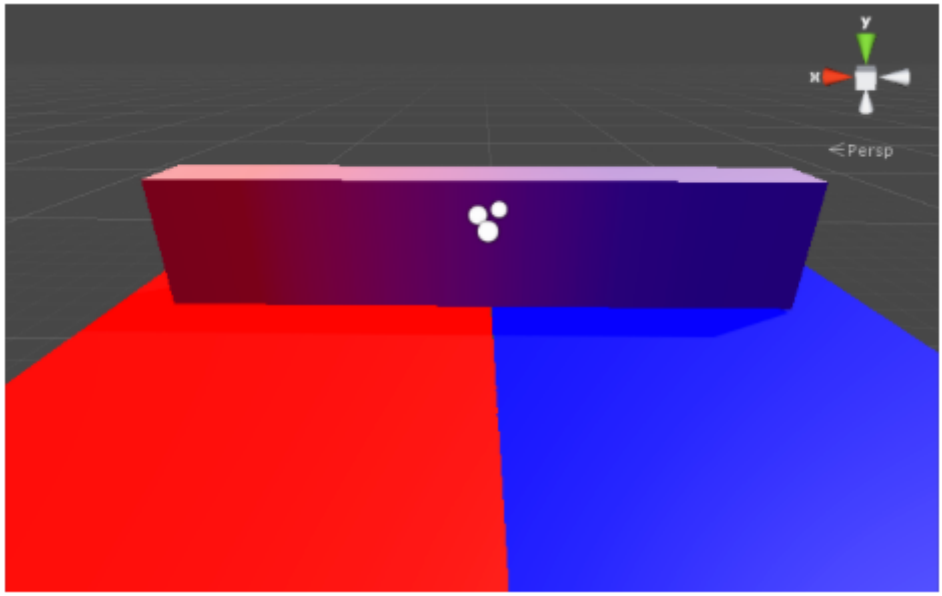
Note: The final resolution on each axis must be a power of two, and the maximum value of the resolution is 32.

Probe Position Mode specifies the relative position of an interpolated Light Probe to a cell center. This option may be useful in situations when some of the interpolated Light Probes pass through walls or other geometries and cause light leaking. The example below shows the difference between **Cell Corner** and **Cell Center** in a 2D view, using a 4x4 grid resolution:

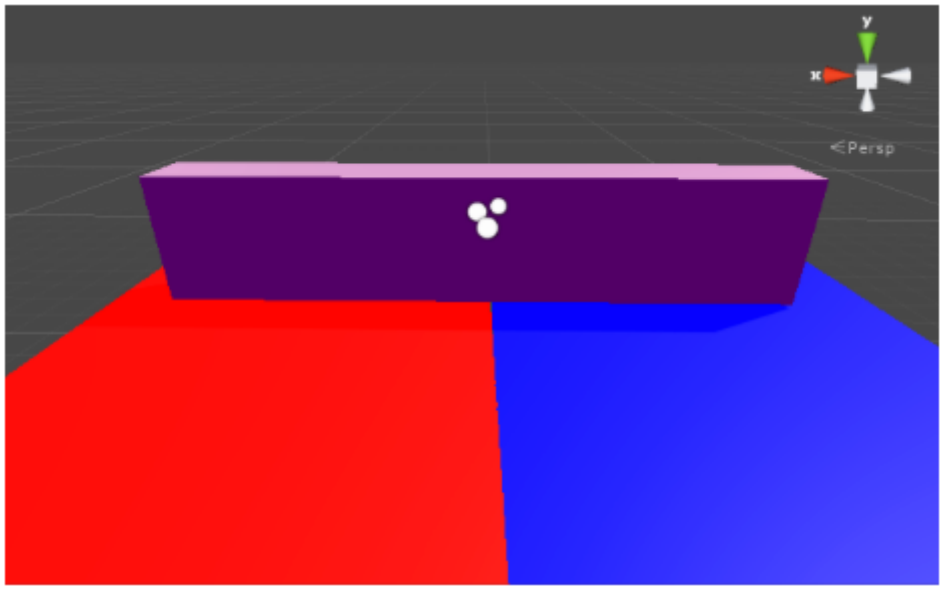


Images for comparison

A simple **Mesh** Renderer using a Standard Shader:

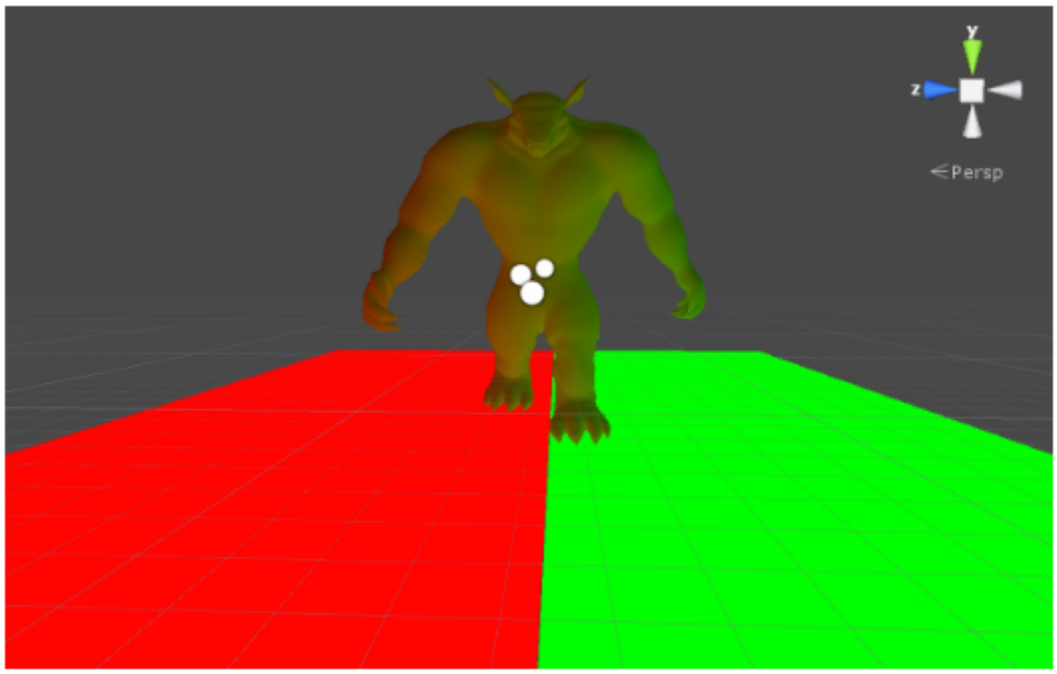


With Light Probe Proxy Volume (resolution: 4x1x1)

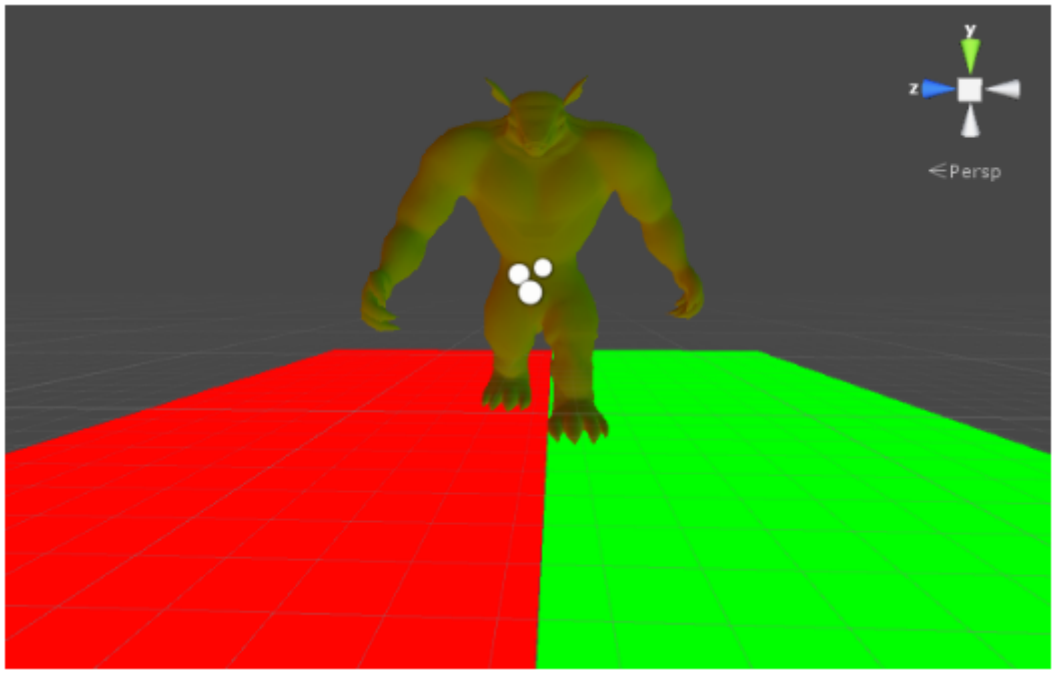


Without Light Probe Proxy Volume

A skinned **Mesh Renderer** using a Standard Shader:



With Light Probe Proxy Volume (resolution: 2x2x2)



Without Light Probe Proxy Volume

Particle System sample Shader using the ShadeSHPerPixel function

```
Shader "Particles/AdditiveLPPV" {
Properties {
    _MainTex ("Particle Texture", 2D) = "white" {}
    _TintColor ("Tint Color", Color) = (0.5,0.5,0.5,0.5)
}

Category {
    Tags { "Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transpar
    Blend SrcAlpha One
    ColorMask RGB
    Cull Off Lighting Off ZWrite Off

    SubShader {
        Pass {

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile_particles
            #pragma multi_compile_fog

            // Specify the target
            #pragma target 3.0
```

```
#include "UnityCG.cginc"

// You must include this header to have access to ShadeSHPerPixel
#include "UnityStandardUtils.cginc"

fixed4 _TintColor;
sampler2D _MainTex;

struct appdata_t {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    fixed4 color : COLOR;
    float2 texcoord : TEXCOORD0;
};

struct v2f {
    float4 vertex : SV_POSITION;
    fixed4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    UNITY_FOG_COORDS(1)
    float3 worldPos : TEXCOORD2;
    float3 worldNormal : TEXCOORD3;
};

float4 _MainTex_ST;

v2f vert (appdata_t v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.worldNormal = UnityObjectToWorldNormal(v.normal);
    o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
    o.color = v.color;
    o.texcoord = TRANSFORM_TEX(v.texcoord,_MainTex);
    UNITY_TRANSFER_FOG(o,o.vertex);
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    half3 currentAmbient = half3(0, 0, 0);
    half3 ambient = ShadeSHPerPixel(i.worldNormal, currentAmbient, i
    fixed4 col = _TintColor * i.color * tex2D(_MainTex, i.texcoord);
    col.xyz += ambient;
    UNITY_APPLY_FOG_COLOR(i.fogCoord, col, fixed4(0,0,0,0)); // fog
    return col;
```

```
                }
                ENDCG
            }
        }
    }
}
```

Hardware requirements

The component requires at least Shader Model 4 graphics hardware and API support, including support for 3D Textures with 32-bit floating-point format and linear filtering.

To work correctly, the Scene needs to contain Light Probes via **Light Probe Group** components. If a requirement is not fulfilled, the Renderer or Light Probe Proxy Volume component **Inspector** displays a warning message.

Reflection probes

Leave feedback

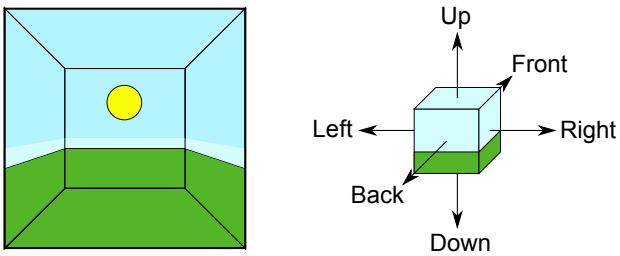
CG films and animations commonly feature highly realistic reflections, which are important for giving a sense of "connectedness" among the objects in the **scene**. However, the accuracy of these reflections comes with a high cost in processor time and while this is not a problem for films, it severely limits the use of reflective objects in realtime games.

Traditionally, games have used a technique called *reflection mapping* to simulate reflections from objects while keeping the processing overhead to an acceptable level. This technique assumes that all reflective objects in the scene can "see" (and therefore reflect) the exact same surroundings. This works quite well for the game's main character (a shiny car, say) if it is in open space but is unconvincing when the character passes into different surroundings; it looks strange if a car drives into a tunnel but the sky is still visibly reflected in its windows.

Unity improves on basic reflection mapping through the use of **Reflection Probes**, which allow the visual environment to be sampled at strategic points in the scene. You should generally place them at every point where the appearance of a reflective object would change noticeably (eg, tunnels, areas near buildings and places where the ground colour changes). When a reflective object passes near to a probe, the reflection sampled by the probe can be used for the object's reflection map. Furthermore, when several probes are nearby, Unity can interpolate between them to allow for gradual changes in reflections. Thus, the use of reflection probes can create quite convincing reflections with an acceptable processing overhead.

How Reflection Probes Work

The visual environment for a point in the scene can be represented by a cubemap. This is conceptually like a box with flat images of the view from six directions (up, down, left, right, forward and backward) painted on its interior surfaces.



Inside surfaces of a skybox cubemap (front face removed)

For an object to show the reflections, its **shader** must have access to the images representing the cubemap. Each point of the object's surface can "see" a small area of cubemap in the direction the surface faces (ie, the direction of the surface normal vector). The shader uses the colour of the cubemap at this point in calculating what colour the object's surface should be; a mirror material might reflect the colour exactly while a shiny car might fade and tint it somewhat.

As mentioned above, traditional reflection mapping makes use of only a single cubemap to represent the surroundings for the whole scene. The cubemap can be painted by an artist or it can be obtained by taking six "snapshots" from a point in the scene, with one shot for each cube face. Reflection probes improve on this by allowing you to set up many predefined points in the scene where cubemap snapshots can be taken. You can therefore record the surrounding view at any point in the scene where the reflections differ noticeably.

In addition to its view point, a probe also has a zone of effect defined by an invisible box shape in the scene. A reflective object that passes within a probe's zone has its reflection cubemap supplied temporarily by that probe.

As the object moves from one zone to another, the cubemap changes accordingly.

Types of Reflection Probe

Leave feedback

Reflection probes come in three basic types as chosen by the *Type* property in the **inspector** (see the component reference page for further details).

Baked probes store a reflection **cubemap** generated ("baked") within the editor. You can trigger the baking by clicking either the *Bake* button at the bottom of the Reflection Probe inspector or the *Build* button in the Lighting window. If you have *Auto* enabled in the Lighting window then baked probes will be updated automatically as you place objects in the **Scene view**. The reflection from a baked probe can only show objects marked as *Reflection Probe Static* in the inspector. This indicates to Unity that the objects will not move at runtime.
Realtime probes create the cubemap at runtime in the player rather than the editor. This means that the reflections are not limited to static objects and can be updated in real time to show changes in the **scene**. However, it takes considerable processing time to refresh the view of a probe so it is wise to manage the updates carefully. Unity allows you to trigger updates from a script so you can control exactly when they happen. Also, there is an option to apply *timeslicing* to probe updates so that they can take place gradually over a few frames.
A **Custom** probe type is also available. These probes let you bake the view in the editor, as with Baked probes, but you can also supply a custom cubemap for the reflections. Custom probes cannot be updated at runtime.
The three types are explained in detail below.

Baked and Custom Reflection Probes

A **Baked** reflection probe is one whose reflection cubemap is captured in the Unity editor and stored for subsequent usage in the player (see the Reflection Probes Introduction for further information). Once the capture process is complete, the reflections are "frozen" and so baked probes can't react to runtime changes in the scene caused by moving objects. However, they come with a much lower processing overhead than Realtime probes (which do react to changes) and are acceptable for many purposes. For example, if there is only a single moving reflective object then it need only reflect its static surroundings.

Using Baked probes

You should set the probe's *Type* property to *Baked* or *Custom* in order to make it behave as a baked probe (see below for the additional features offered by Custom probes).

The reflections captured by baked probes can only include scene objects marked as *Reflection Probe Static* (using the *Static* menu at the top left of the inspector panel for all objects). You can further refine the objects that get included in the reflection cubemap using the *Culling Mask* and *Clipping Planes* properties, which work the same way as for a Camera (the probe is essentially like a camera that is rotated to view each of the six cubemap faces).

When the *Auto* option is switched on (from the Lighting window, the baked reflections will update automatically as you position objects in the scene. If you are not making use of auto baking then you will need to click the *Bake* button in the Reflection Probe inspector to update the probes. (The *Build* button in the Lighting window will also trigger the probes to update.)

Whether you use auto or manual baking, the bake process will take place asynchronously while you continue to work in the editor. However, if you move any static objects, change their materials or otherwise alter their visual appearance then the baking process will be restarted.

Custom Probes

By default, Custom probes work the same way as Baked probes but they also have additional options that change this behaviour.

The *Dynamic Objects* property on a custom probe's inspector allows objects that are not marked as *Reflection Probe Static* to be included in the reflection cubemap. Note, however, that the positions of these objects are still "frozen" in the reflection at the time of baking.

The *Cubemap* property allows you to assign your own cubemap to the probe and therefore make it completely independent of what it can "see" from its view point. You could use this, say, to set a **skybox** or a cubemap generated from your 3D modelling app as the source for reflections.

Realtime Probes

Baked probes are useful for many purposes and have good runtime performance but they have the disadvantage of not updating live within the player. This means that objects can move around in the scene without their reflections moving along with them. In cases where this is too limiting, you can use **Realtime** probes, which update the reflection cubemap at runtime. This effect comes with a higher processing overhead but offers greater realism.

Using Realtime probes

To enable a probe to update at runtime, you should set its *Type* property to *Realtime* in the Reflection Probe inspector. You don't need to mark objects as *Reflection Probe Static* to capture their reflections (as you would with a baked probe). However, you can selectively exclude objects from the reflection cubemap using the *Culling Mask* and *Clipping Planes* properties, which work the same way as for a Camera (the probe is essentially like a camera that is rotated to view each of the six cubemap faces).

In the editor, realtime probes have much the same workflow as baked probes, although they tend to render more quickly. When the *Auto* option is switched on (from the Lighting window, the reflections will update automatically as you position objects in the scene. If you are not making use of auto baking then you will need to click the *Bake* button in the Reflection Probe inspector to update the probes. (The *Build* button in the Lighting window will also trigger the probes to update.)

Whether you use auto or manual baking, the bake process will take place asynchronously while you continue to work in the editor. However, if you move any static objects, change their materials or otherwise alter their visual appearance then the baking process will be restarted.

Note: Currently, realtime probes will only update their reflections in the Scene view when *Reflection Probe Static* objects are moved or change their appearance. This means that moving dynamic objects will not cause an update even though those objects appear in the reflection. You should choose the *Bake Reflection Probes* option from the *Build* button popup on the Lighting window to update reflections when a dynamic object is changed.

Using Reflection Probes

Leave feedback

You can add the Reflection Probe component to any object in a **Scene** but it's standard to add each probe to a separate empty GameObject. The usual workflow is:

Create a new empty GameObject (menu: **GameObject** > **Create Empty**) and then add the Reflection Probe component to it (menu: **Component** > **Rendering** > **Reflection Probe**). Alternatively, if you already have a probe in the scene you will probably find it easier to duplicate that instead (menu: **Edit** > **Duplicate**).
Place the new probe in the desired location and set its **Offset** point and the size of its zone of effect.
Optionally set other properties on the probe to customise its behaviour.
Continue adding probes until all required locations have been assigned.
To see the reflections, you will also need at least one reflective object in the scene. A simple test object can be created as follows:

Add a primitive object such as a Sphere to the scene (menu: **GameObject** > **3D Object** > **Sphere**).
Create a new material (menu: **Assets** > **Create** > **Material**) and leave the default **Standard** shader in place.
Make the material reflective by setting both the **Metallic** and **Smoothness** properties to **1.0**.
Drag the newly-created material onto the sphere object to assign it.
The sphere can now show the reflections obtained from the probes. A simple arrangement with a single probe is enough to see the basic effect of the reflections.

Finally, the probes must be baked before the reflections become visible. If you have the **Auto Generate** option enabled in the Lighting window (this is the default setting) then the reflections will update as you position or change objects in the scene, although the response is not instantaneous. If you disable auto baking then you must click the **Bake** button in the Reflection Probe **inspector** to update the probes. The main reason for disabling auto baking is that the baking process can take quite some time for a complicated scene with many probes.

Positioning probes

The position of a probe is primarily determined by the position of its GameObject and so you can simply drag the object to the desired location. Having done this, you should set the probe's zone of effect; this is an axis-aligned box shape whose dimensions are set by the **Box Size** property. You can set the size values directly or enable the size editing mode in the inspector and drag the sides of the box in the **Scene view** (see the Reflection Probe component page for details). The zones of the full set of probes should collectively cover all areas of the scene where a reflective object might pass.

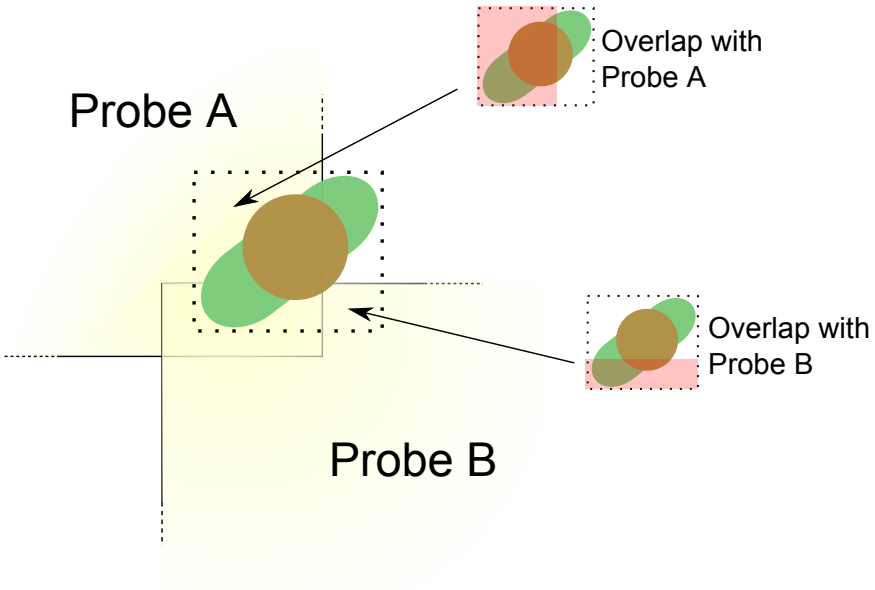
You should place probes close to any large objects in the scene that would be reflected noticeably. Areas around the centres and corners of walls are good candidate locations for probes. Smaller objects might require probes close by if they have a strong visual effect. For example, you would probably want the flames of a campfire to be reflected even if the object itself is small and otherwise insignificant.

When you have probes in all the appropriate places, you then need to define the zone of effect for each probe, which you can do using the **Box Size** property as mentioned above. A wall might need just a single probe zone along most of its length (at least if it has a fairly uniform appearance) but the zone might be relatively narrow in the direction perpendicular to the wall; this would imply that the wall is only reflected by objects that are fairly close to it. An open space whose appearance varies little from place to place can often be covered by a single probe. Note that a probe's zone is aligned to the main world axes (X, Y and Z) and can't be rotated. This means that sometimes a group of probes might be needed along a uniform wall if it is not axis-aligned.

By default, a probe's zone of effect is centred on its view point but this may not be the ideal position for capturing the reflection **cubemap**. For example, the probe zone for a very high wall might extend some distance from the wall but you might want the reflection to be captured from a point close to it rather than the zone's centre. You can optionally add an offset to view point using the **Box Offset** property (ie, the offset is the position in the GameObject's local space that the probe's cubemap view is generated from). Using this, you can easily place the view point anywhere within the zone of effect or indeed outside the zone altogether.

Overlapping probe zones

It would be very difficult to position the zones of neighbouring reflection probes without them overlapping and fortunately, it is not necessary to do so. However, this leaves the issue of choosing which probe to use in the overlap areas. By default, Unity calculates the intersection between the reflective object's bounding box and each of the overlapping probe zones; the zone which has the largest volume of intersection with the bounding box is the one that will be selected.



Probe A is selected since its intersection with the object is larger

You can modify the calculation using the probes' **Importance** properties. Probes with a higher importance value have priority over those of lower importance within overlap zones. This is useful, say, if you have a small probe zone that is contained completely inside a larger zone (ie, the intersection of the character's bounding box with the enclosing zone might always be larger and so the small zone would never be used).

Blending

To enable Reflection Probe blending, navigate to **Graphic Settings** > **Tier settings**. With blending enabled, Unity will gradually fade out one probe's cubemap while fading in the other's as the reflective object passes from one zone to the other. This gradual transition avoids the situation where a distinctive object suddenly "pops" into the reflection as an object crosses the zone boundary.

Blending is controlled using the **Reflection Probes** property of the Mesh Renderer component. Four blending options are available:

Off - Reflection probe blending is disabled. Only the **skybox** will be used for reflection

Blend Probes - Blends only adjacent probes and ignores the skybox. You should use this for objects that are "indoors" or in covered parts of the scene (eg, caves and tunnels) since the sky is not visible from these place and so should never appear in the reflections.

Blend Probes and Skybox - Works like *Blend Probes* but also allows the skybox to be used in the blending. You should use this option for objects in the open air, where the sky would always be visible.
Simple - Disables blending between probes when there are two overlapping reflection probe volumes.
When probes have equal **Importance** values, the blending weight for a given probe zone is calculated by dividing its intersection (volume) with the object's bounding box by the sum of all probes' intersections with the box. For example, if the box intersects probe A's zone by 1.0 cubic units and intersects probe B's zone by 2.0 cubic units then the blending values will be:

Probe A: 1.0 / (1.0 + 2.0) = 0.33
Probe B: 2.0 / (1.0 + 2.0) = 0.67
In other words, the blend will incorporate 33% of probe A's reflection and 67% of probe B's reflection.

The calculation must be handled slightly differently in the case where one probe is entirely contained within the other, since the inner zone overlaps entirely with the outer. If the object's bounding box is entirely within the inner zone then that zone's blending weight is 1.0 (ie, the outer zone is not used at all). When the object is partially outside the inner zone, the intersection volume of its bounding box with the inner zone is divided by the total volume of the box. For example, if the intersection volume is 1.0 cubic units and the bounding box's volume is 4.0 cubic units, then the blending weight of the inner probe will be 1.0 / 4.0 = 0.25. This value is then subtracted from 1.0 to get the weight for the outer probe which in this case will be 0.75.

When one probe involved in the blend has a higher **Importance** value than another, the more important probe overrides the other in the usual way.

Updated in 5.6

2017–06–06 Page published with limited editorial review

Advanced Reflection Probe Features

Leave feedback

Two further features which can improve the visual realism obtained from Reflection Probes are described below: **Interreflections** and **Box Projection**.

Interreflections

You may have seen a situation where two mirrors are placed fairly close together and facing each other. Both mirrors reflect not only the mirror opposite but also the reflections produced by that mirror. The result is an endless progression of reflections between the two; reflection between objects like this are known as **Interreflections**.

Reflection probes create the **cubemap** by taking a snapshot of the view from their position. However, with a single snapshot, the view cannot show interreflections and so additional snapshots must be taken for each stage in the interreflection sequence.

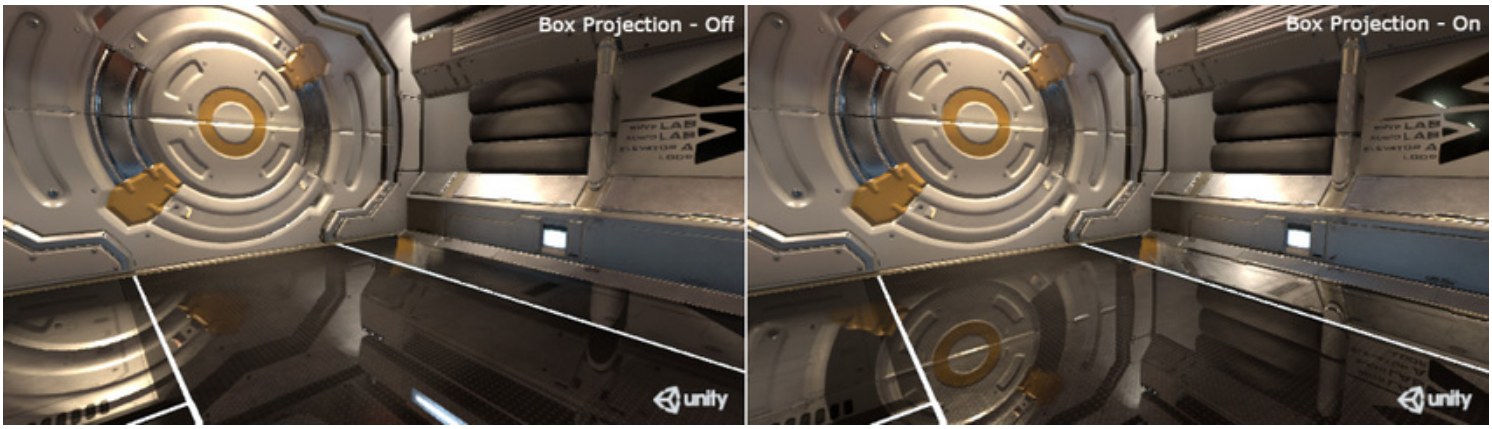
The number of times that a reflection can "bounce" back and forth between two objects is controlled in the Lighting window; go to **Environment** > **Environment Reflections** and edit the **Bounces** property. This is set globally for all probes, rather than individually for each probe. With a reflection bounce count of 1, reflective objects viewed by a probe are shown as black. With a count of 2, the first level of interreflection are visible, with a count of 3, the first two levels will be visible, and so on.

Note that the reflection bounce count also equals the number of times the probe must be baked with a corresponding increase in the time required to complete the full bake. You should therefore set the count higher than one only when you know that reflective objects will be clearly visible in one or more probes.

Box projection

Normally, the reflection cubemap is assumed to be at an infinite distance from any given object. Different angles of the cubemap will be visible as the object turns but it is not possible for the object to move closer or farther away from the reflected surroundings. This often works very well for outdoor **scenes** but its limitations show in an indoor scene; the interior walls of a room are clearly not an infinite distance away and the reflection of a wall should get larger the closer the object gets to it.

The **Box Projection** option allows you to create a reflection cubemap at a finite distance from the probe, thus allowing objects to show different-sized reflections according to their distance from the cubemap's walls. The size of the surrounding cubemap is determined by the probes zone of effect, as determined by its **Box Size** property. For example, with a probe that reflects the interior of a room, you should set the size to match the dimensions of the room. Globally, you can enable **Box Projection** from **Project Settings** > **Graphics** > **Tier Settings**, but the option can be turned off from the Reflection Probe inspector for specific Reflection Probes when infinite projection is desired.



The parallax issue is fixed by using Box Projection option

Reflection probe performance and optimisation

Leave feedback

Rendering a **reflection probe**'s **cubemap** takes a significant amount of processor time for a number of reasons:

Each of the six cubemap faces must be rendered separately using a "**camera**" at the probe's origin.

The probes will need to be rendered a separate time for each reflection bounce level (see documentation on Advanced Reflection Probes for further details).

The various mipmap levels of the cubemap must undergo a blurring process to allow for glossy reflections.

The time taken to render the probes affects the baking workflow in the editor and, more importantly, runtime performance of the player. Below are some tips for keeping the performance impact of reflection probes to a minimum.

General tips

The following issues affect both offline baking *and* runtime performance.

Resolution

The higher the resolution of a cubemap, the greater its rendering time will be. You can optimise probes by setting lower resolutions in places where the reflection detail is less important (for example, if a reflective object is small and/or distant then it will naturally show less detail). Higher resolutions should still be used wherever the detail will be noticeable.

Culling Mask

A standard technique to improve a normal camera's performance is to use the **Culling Mask** property to avoid rendering insignificant objects; the technique works equally well for reflection probes. For example, if your **Scene** contains many small objects (eg, rocks and plants) you might consider putting them all on the same layer and then using the culling mask to avoid rendering them in the reflection.

Texture compression

To optimize the rendering time and lower the GPU memory consumption, use texture compression. To control texture compression for baked Reflection Probes, open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**), navigate to **Environmental Lighting** > **Reflections** and use the **Compression** drop-down menu. Note that real-time Reflection Probes are not compressed in memory, and their size in memory depends on Resolution and HDR settings. Because of this, sampling real-time Reflection Probes is usually more resource-intensive than sampling baked Reflection Probes.

Real-time probe optimisation

The rendering overhead is generally more significant for real-time probes than for those baked in the editor. Updates are potentially quite frequent and this can have an impact on framerate if not managed correctly. With this in mind, real-time probes provide the following properties to let you handle probe rendering as efficiently as possible:

Refresh Mode

The **Refresh Mode** lets you choose when the probe will update. The most resource-intensive option in terms of processor time is **Every Frame**. This gives the most frequent updates with minimal programming effort but you may encounter performance problems if you use this mode for all probes.

If the mode is set to **On Awake**, the probe will be updated at runtime but only once at the start of the scene. This is useful if the scene (or part of it) is set up at run-time but does not change during its lifetime.

The final mode, **Via Scripting**, lets you control probe updates from a script. Although some effort is involved in coding the script, this approach does allow for useful optimisations. For example, you might update a probe according to the apparent size of passing objects (ie, small objects or large objects at a distance are not worth an update).

Time Slicing

When the **Refresh Mode** described above is set to **Every Frame** the processing load can be considerable. **Time Slicing** allows you to spread the cost of updates over several frames and thereby reduce the load at any given time. This property has three different options:

All Faces at Once will cause the six cubemap faces to be rendered immediately (on the same frame) but then the blurring operation for each of the six first level mipmaps will take place on separate frames. The remaining mipmaps will then be blurred on a single frame and the results copied to the cubemap on another frame. The full update therefore takes nine frames to complete.

Individual Faces works the same way as **All Faces at Once** except that the initial rendering of each cubemap face takes place on its own frame (instead of all six on the first frame). The full update takes fourteen frames to complete; this option has the lowest impact on framerate but the relative long update time might be noticeable when, say, lighting conditions change abruptly (eg, a lamp is suddenly switched on).

No Time Slicing disables the time slicing operation completely and so each probe update takes place within a single frame. This ensures that the reflections are synchronised exactly with the appearance of surrounding objects but the processing cost can be prohibitive.

As with the other optimisations, you should consider using the lower-quality options in places where reflections are less important and save the **No Time Slicing** option for places where the detail will be noticed.

Updated in 5.6

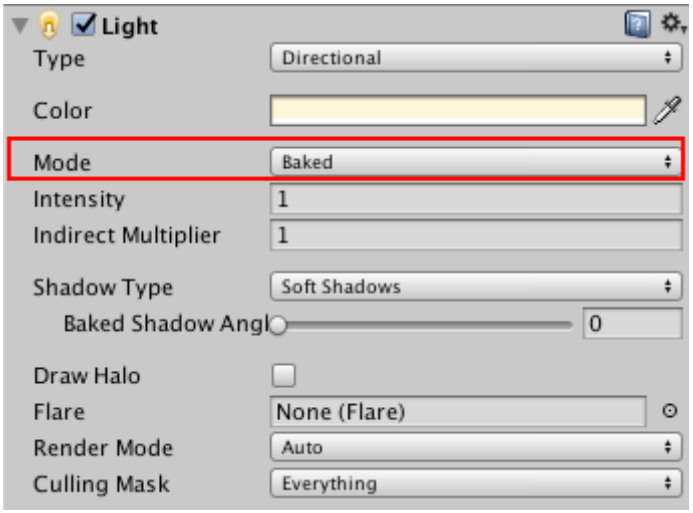
2017–06–06 Page published with limited editorial review

Lighting Modes

Leave feedback

This page assumes you have already read documentation on lighting in Unity.

To control lighting precomputation and composition in Unity, you need to assign a **Light Mode** to a Light. This **Light Mode** defines the Light's intended use. To assign a Light Mode, select the Light in your Scene and, in the Light Inspector window, select **Mode**.



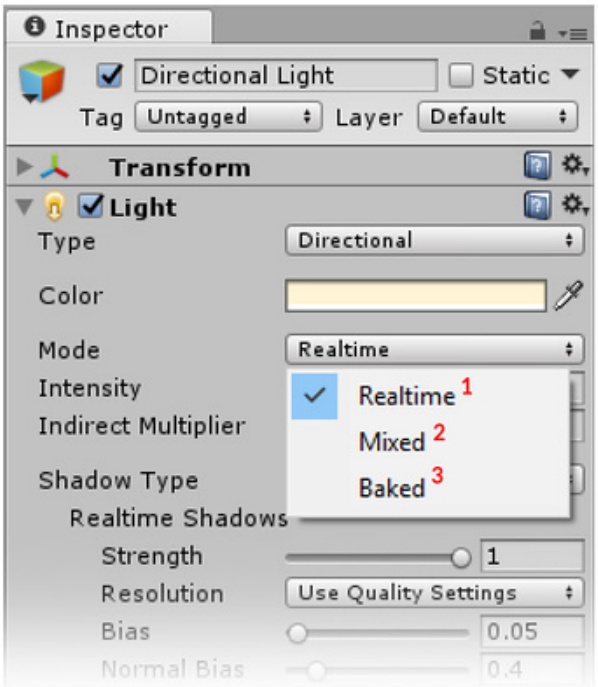
The Modes and their possible mappings are:

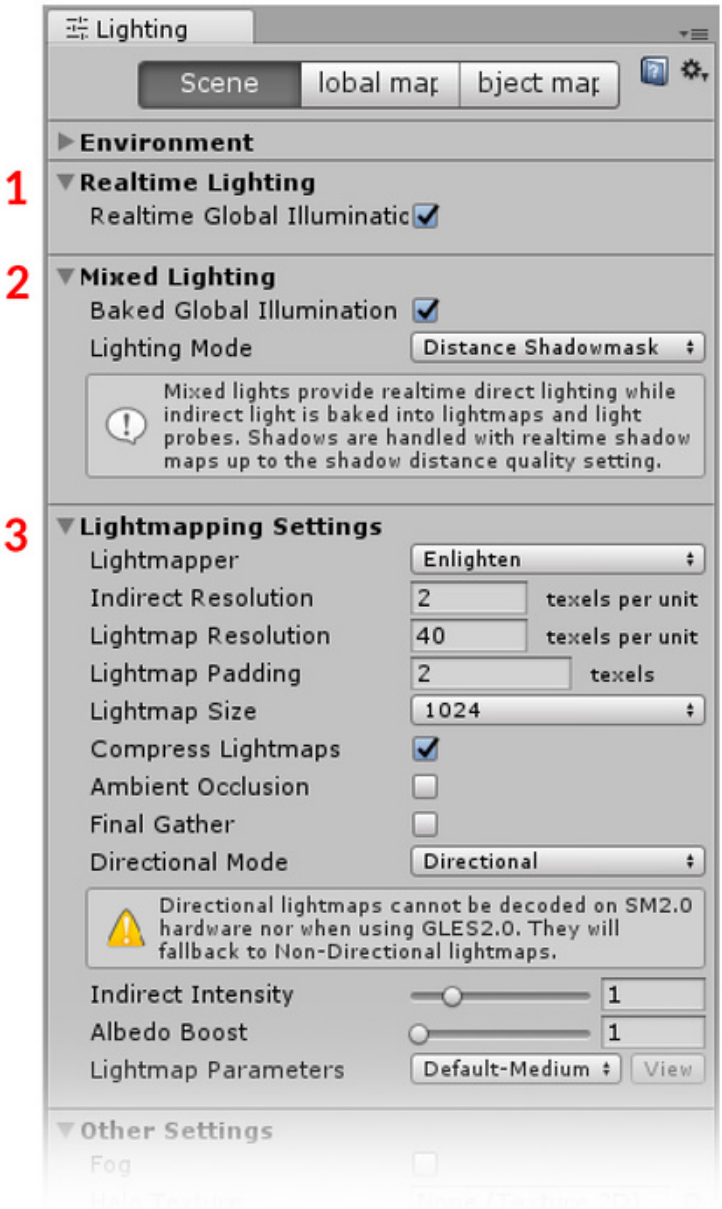
Realtime
Mixed - Mixed Lights have their own sub-modes. Set these in the Lighting window:
Baked Indirect
Shadowmask
Distance Shadowmask
Subtractive
Baked(LightMode-Baked)
For more information, see the Reference card for Light Modes.

The Modes are listed above in the order of least to most light path precomputations required (See *How Modes work*, below). Note that this order does not necessarily correlate with the amount of time the actual precomputation requires.

How Modes work

Each **Mode** in the Light Inspector window corresponds to a group of settings in the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings** > **Scene**).





The available **Light Modes** in the Light component (left), and the corresponding settings available for those modes in the Lighting Scene window (right).

Light Inspector	Lighting window	Function
Realtime	**Realtime Lighting**	Unity calculates and updates the lighting of **Realtime** Lights every frame during run time. No Realtime Lights are precomputed.
Mixed	**Mixed Lighting**	Unity can calculate some properties of **Mixed** Lights during run time, but only within strong limitations. Some Mixed Lights are precomputed.
Baked	**Lightmapping Settings**	Unity pre-calculates the illumination from **Baked** Lights before run time, and does not include them in any run-time lighting calculations. All **Baked Lights** are precomputed.

Use these settings to adjust each mode. The adjustments you make apply to all Lights with that Mode assigned to them. For example, if you open the Lighting window, navigate to the **Realtime Lighting Settings** and tick **Realtime Global Illumination**, all Lights that have their **Mode** set to **Realtime** mode use **Realtime Global Illumination**.

Precomputation yields two sets of results:

Unity stores results for static **GameObjects** as Texture atlases in UV Texture coordinate space. Unity provides several settings to control this layouting.

Light Probes store a representation of light in empty space as seen from their particular position. Dynamic GameObjects moving through this portion of empty space use this information to receive illumination from the precomputed lighting.

2017–06–08 Page published with limited editorial review

Updated in 5.6

Lighting: Technical information and terminology

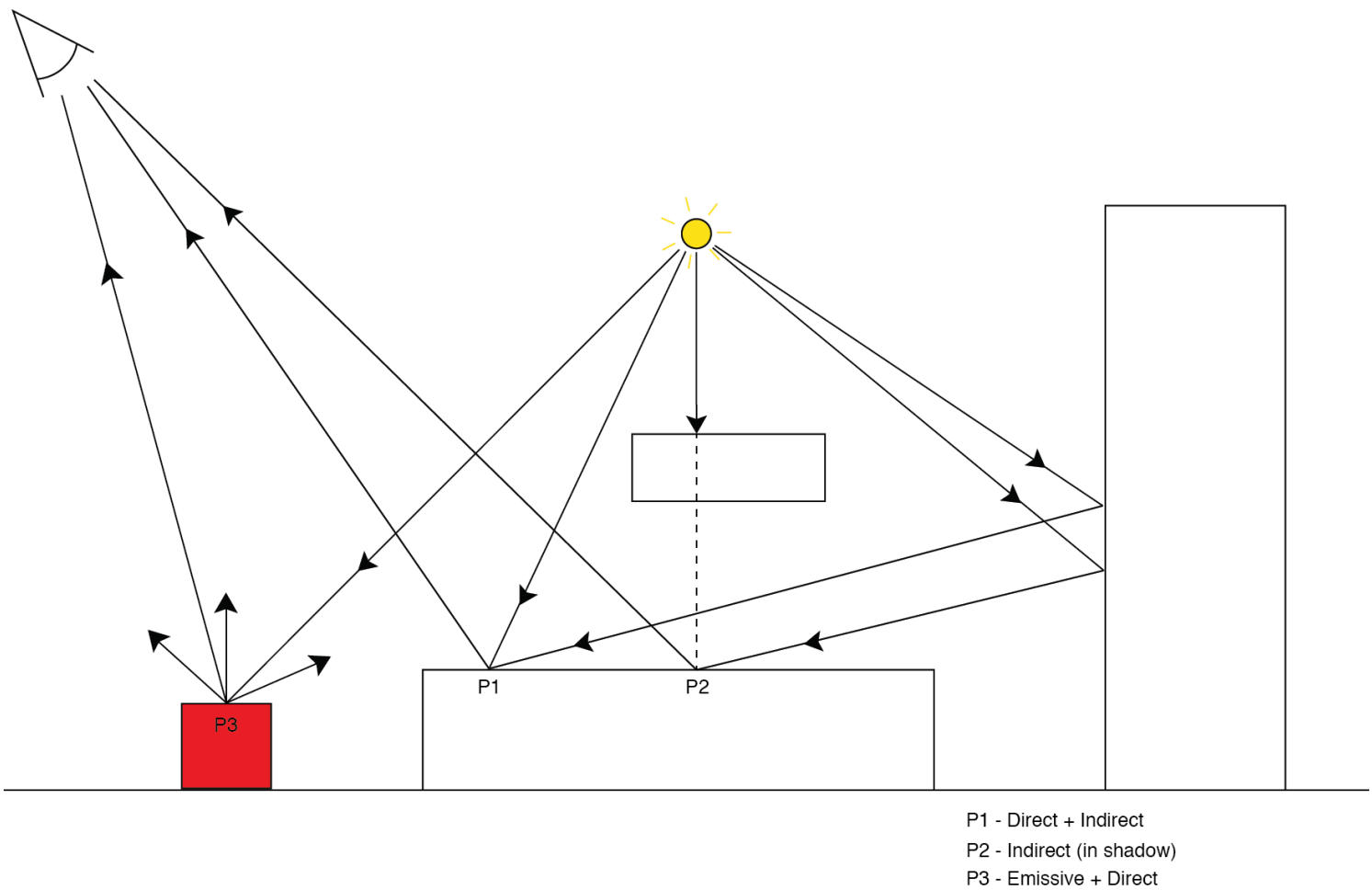
Leave feedback

Surface: All triangles from all meshes in a **Scene** together are called the **surface** of a **scene**. A surface point is a point within any triangle defined for the **Scene**.

Emitted Light: This is light that is emitted directly onto the surface of the Scene.

Direct Light: This is light that is emitted, hits the surface of the Scene and is then reflected into a sensor (for example, the eye's retina or a camera). A Light's direct contribution is any direct light that arrives at a sensor from that Light.

Indirect Light: This is light that is emitted, hits the surface of the Scene at least two times and is ultimately reflected into a sensor. A Light's indirect contribution is any indirect light that arrives at a sensor from that Light.



Reflection of simulated light in a Scene

Rough surfaces scatter incoming light in many directions, illuminating surfaces that are not directly lit from light sources. The rougher the surfaces in a scene, the brighter such shadowed areas will appear. In the past, this effect was approximated by defining one additional ambient light color which was simply added to the result of direct lighting, so that surfaces in shadows would not appear completely black. More sophisticated approximations use a gradient to simulate different ambient colors depending on the orientation of the surface, or even spherical harmonics to have even more complex ambient lighting.

Smooth or glossy surfaces reflect most of the incoming light in predictable directions, leading to visible highlights on materials. The extreme case of a glossy surface is a mirror, where all the incoming light from one direction gets reflected into exactly one other direction. A variation of glossy reflections are translucent materials, that can also refract incoming light when it enters and leaves the material again.

In the case of indirect lighting, a light path has at least two interactions with the Scene's surface. These interactions can be a combination of glossy and/or rough surface reflections. For example, glossy reflections/refractions hitting a rough surface display patterns of focused light and darkness visible from all viewing directions, which are called **caustics**. Rough reflections hitting another rough surface are usually referred to as **ambient lighting**.

Due to the nature of light being reflected multiple times on the surface of a scene, a correct solution needs to take the entire scene with all its surface material properties and light interactions for all relevant light paths into account. Hence the term Global Illumination.

Solving the problem

Ray tracing is a very elegant way of solving this problem in computer graphics, as it tries to simulate what actually happens in the real world by following ray paths through the scene. The movie industry has entirely moved to ray tracing at this point for generating their images.

Unfortunately, ray tracing is still too slow to be used in most real time graphics, where rasterization is the standard method of generating images. Unlike ray tracing, rasterization cannot follow arbitrary light paths through the scene. In fact, a rasterizer can only ever calculate one segment of a light path. This is why lighting in real time graphics gets complicated.

Since rasterizers cannot follow light rays, real time lighting concentrates on the parts of lighting with the most visible impact. These are emission, and more commonly, direct lighting. Even in this case the light path already consists of two segments - one from the **camera** to the surface, and one from the surface to the light.

The first segment is the view rendered from the camera position. In order to calculate the second segment, techniques like shadow mapping are used. Since shadow maps are specific to each shadow-casting Light, a unique shadow map must be generated for each one. The more shadow-casting Lights there are, the more shadow maps need to be generated. Depending on the number of Lights there are, required **rendering** time can quickly become too long. Another drawback of shadow maps is their limited resolution. This leads to blocky shadows. Therefore, shadow maps present both an image quality issue due to the limited resolution, and a performance issue due to the memory requirements to store the shadow maps and the time it takes to generate them every frame.

Unlike offline rendering, games have certain hard limits on how much time they can spend rendering a frame. For instance, VR applications have 11.11 milliseconds to draw a frame, in order to achieve 90 **frames per second** (FPS). Games relying on fast player reactions have 16.66ms to draw a frame in order to hit 60 **FPS**. Games that target 30 **FPS** have 33.33ms. These times must also include calculations for the rest of the application or game, such as AI and physics. It is therefore important to make everything as efficient as possible to get the most out of the system. All rendering must occur within less than the time between frames.

Summary

To recap, the two major issues that need to be addressed are:

How to deal with the performance penalty caused by calculating shadows for direct lighting.

How to deal with indirect lighting (note: in the context of real time graphics, Global Illumination is synonymous with indirect lighting, even though the actual meaning encompasses direct lighting as well).

2017–06–08 Page published with limited editorial review

Light Modes added in 5.6

Real-time lighting

Leave feedback

Real-time Lights are Light components which have their **Mode** property set to **Realtime**.

Use **Realtime** mode for Lights that need to change their properties or which are spawned via **scripts** during gameplay. Unity calculates and updates the lighting of these Lights every frame at run time. They can change in response to actions taken by the player, or events which take place in the **Scene**. For example, you can set them to switch on and off (like a flickering light), change their Transforms (like a torch being carried through a dark room), or change their visual properties, like their color and intensity.

Real-time Lights illuminate and cast realistic shadows on both static and dynamic GameObjects. They cast shadows up to the **Shadow Distance** (defined in **Edit** > **Project Settings** > **Quality**).

You can also combine **real-time Lights** with **Realtime Global Illumination** (**Realtime GI**), so that they contribute indirect lighting to static and dynamic **GameObjects**.

Using real-time lighting with Realtime GI

The combination of real-time lighting with **Realtime GI** is the most flexible and realistic lighting option in Unity. To enable Realtime GI, open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**) and tick **Realtime Global Illumination**.

When **Realtime GI** is enabled, real-time Lights contribute indirect lighting into the Scene, as well as direct lighting. Use this combination for light sources which change slowly and have a high visual impact on your Scene, such as the sun moving across the sky, or a slowly pulsating light in a closed corridor. You don't need to use **Realtime GI** for Lights that change quickly, or for special effects, because the latency of the system does not make it worth the overhead.

Note that **Realtime GI** uses significant system resources compared to the less complex **Baked GI**. Global Illumination is managed in Unity by a piece of middleware called **Enlighten**, which has its own overheads (system memory and CPU cycles). See documentation on Global Illumination for more information.

Realtime GI is suitable for games targeting mid-level to high-end PC systems, and games targeting current-gen consoles such as the **PS4** and **Xbox One**. Some high-end mobile devices might also be powerful enough to make use of this feature, but you should keep Scenes small and the resolution for real-time light maps low to conserve system resources.

To disable the effect of **Realtime GI** on a specific light, select the Light GameObject and, in the Light component, set the **Indirect Multiplier** to 0. This means that the Light does not contribute any indirect light. To disable **Realtime GI** altogether, open the Lighting window (menu: **Window** > **Lighting** > **Settings**) and untick **Realtime Global Illumination**.

Disadvantages of using real-time lighting with Realtime GI

Increased memory requirements, due to the additional set of low resolution real-time light maps used to store the real-time indirect bounces computed by the Enlighten lighting system.

Increased **shader** calculation requirements, due to sampling of the additional set of real-time light maps and probes used to store the real-time indirect bounces computed by the Enlighten lighting system.

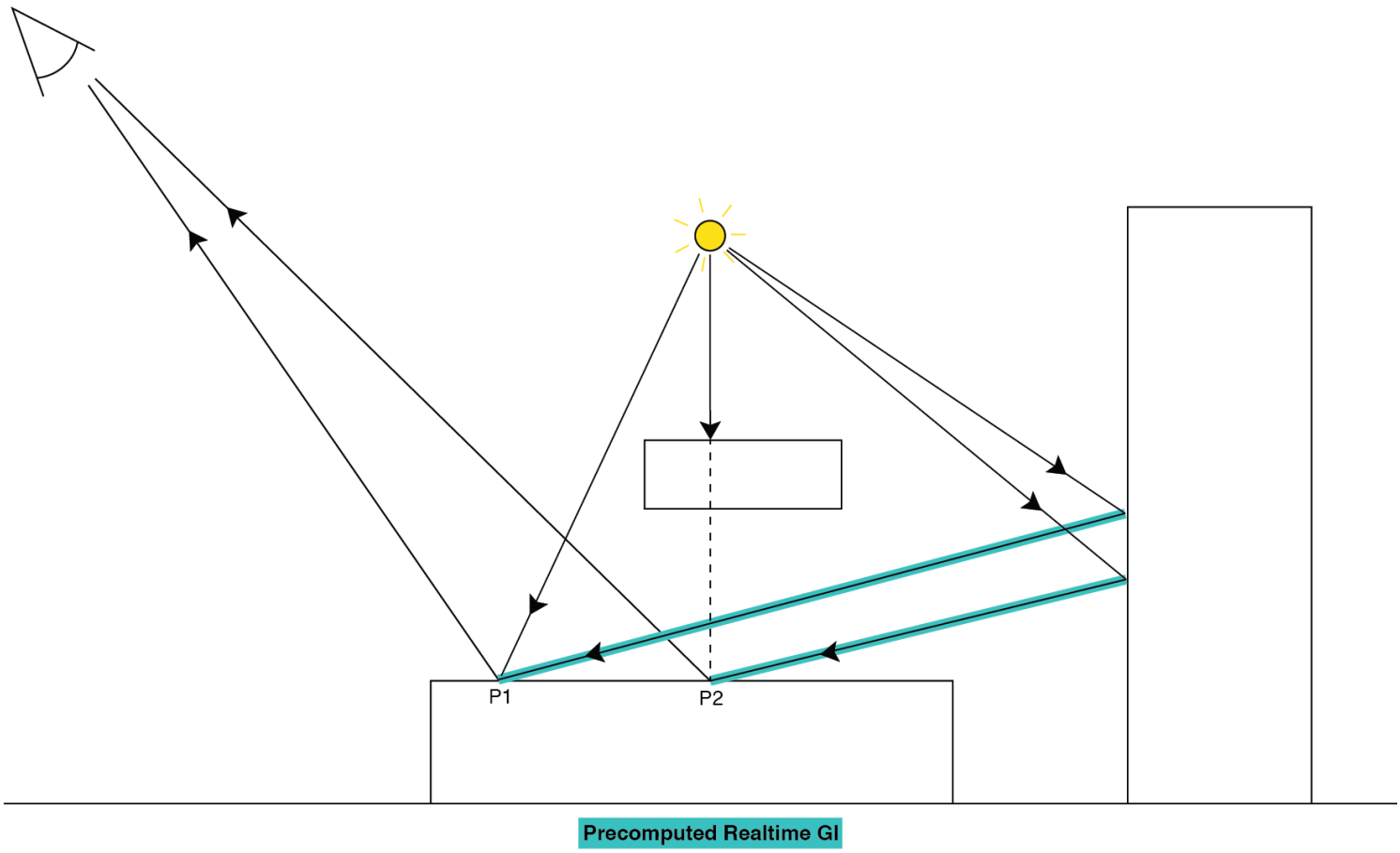
Indirect lighting converges over time, so property changes cannot be too abrupt. Adaptive **HDR** tone mapping might help you hide this; to learn more, see the Unity Post Processing Stack (Asset Store).

Technical details

In the case of real-time Lights (that is, Light components with their **Mode** set to **Realtime**), the last emission (or path segment) from the surface to the Light is not precomputed. This means that Lights can move around the Scene, and change visual properties like color and intensity. See documentation on Using Enlighten in Unity for more information on path segments.

If the Light also casts shadows, both dynamic and static GameObjects in the Scene are rendered into the Light's shadow map. This shadow map is sampled by the Material Shaders of both static and dynamic GameObjects, so that they cast real-time shadows on each other. The **Shadow Distance** (menu: **Edit** > **Project Settings** > **Quality** > **Shadows**) controls the maximum distance at which shadows start to fade out and disappear entirely, which in turn affects performance and image quality.

If **Realtime GI** is not enabled, real-time Lights only calculate direct lighting on dynamic and static GameObjects. If **Realtime GI** is enabled, Unity uses Enlighten to precompute the surface-to-surface light paths for static GameObjects.



Precomputed **Realtime GI** mode: Unity only precomputes surface-to-surface information

The last path segment (that is, the segment from the surface to the Light emitter) is not part of the precomputation. The only information stored is that if the surface is illuminated, then the following surfaces and probes are also illuminated, and the intensities of the various illuminations. There is a separate set of low-resolution real-time light maps, which Enlighten iteratively updates on the CPU at run time with the information of real-time Lights. Because this iterative process is computationally intensive, it is split across several frames. In other words, it takes a couple of frames until the light has fully bounced across the static elements in the Scene, and the real-time light maps and Light Probes have converged to the final result.

For Lights with properties that change slowly (such as a light-emitting sun moving across the sky), this does not pose a problem. However, for Lights with properties that change quickly (such as a flickering lightbulb), the iterative nature of **Realtime GI** may prove unsuitable. Fast property changes do not register significantly with the bounced light system, so there is no point in including them in the calculations.

There are several ways to address this problem. One way is to reduce the real-time light map resolution. Because this results in less calculation at run time, the lighting converges faster. Another option is to increase the CPU Usage setting for the **Realtime**

GI runtime. By dedicating more CPU time, the runtime converges faster. The tradeoff is of course that other systems receive less CPU time to do their work. Whether this is acceptable depends on each individual project. Note that as this is a per-Scene setting, you can dedicate more or less CPU time based on the complexity of each individual Scene in the project.

Even though **Realtime GI** is enabled on a per-Scene basis for all real-time Lights, it is still possible to prevent individual real-time Lights from being considered by **Realtime GI**. To achieve this, set the Light **component**'s **Mode** to **Realtime** and its indirect multiplier to 0, removing all indirect light contributions.

2017–06–08 Page published with limited editorial review

Light Modes added in 5.6

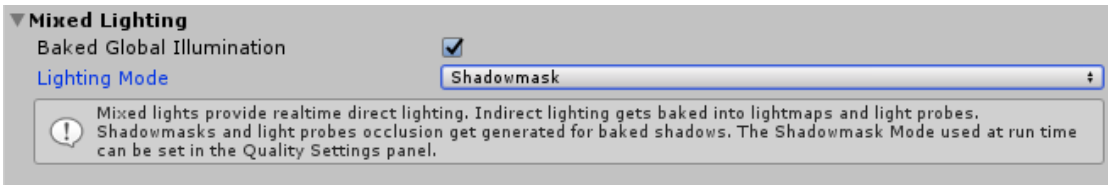
Mixed lighting

Leave feedback

Mixed Lights are Light components which have their **Mode** property set to **Mixed**.

Mixed Lights can change their Transform and visual properties (such as colour or intensity) during run time, but only within strong limitations. They illuminate both static and dynamic **GameObjects**, always provide direct lighting, and can optionally provide indirect lighting. Dynamic **GameObjects** lit by **Mixed** Lights always cast real-time shadows on other dynamic **GameObjects**.

All **Mixed** Lights in a Scene use the same **Mixed Lighting Mode**. To set the **Lighting Mode**, open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**), click the **Scene** tab, and navigate to the **Mixed Lighting** section.



The **Mixed Lighting** section of the Lighting window

The available modes are:

Baked Indirect
Shadowmask
Subtractive

Using Mixed Lights

Mixed lighting is useful for Lights that are not part of gameplay, but which illuminate the static environment (for example, a non-moving sun in the sky). Direct lighting from Mixed Lights is still calculated at run time, so Materials on static Meshes retain their visual fidelity, including full physically based shading (PBS) support.

An example of Mixed Lights

The **Shadowmask** mode's Distance Shadowmask is the most resource-intensive option, but provides the best results: it yields high-quality shadows within the **Shadow Distance** (**Edit** > **Project Settings** > **Quality** > **Shadows**), and baked high-quality shadows beyond. For example, you could create large landscapes with realistic shadows right up to the horizon, as long as the sun does not travel across the sky.

Subtractive mode provides the lowest-quality results: it renders shadows in real time for only one Light, and composites them with baked direct and indirect lighting. Only use this as a fallback solution for target platforms that are unable to use any of the other modes (for example, when the application needs to run on low-end mobile devices, but memory constraints prevent the use of **Shadowmask** or **Distance Shadowmask**).

See the Unity Lighting Modes Reference Card for a condensed comparison of the various modes.

All Mixed Lighting Modes are supported on all platforms. However, there are some rendering limitations:

Subtractive mode falls back to forward rendering (no deferred or light prepass support).

Shadowmask mode falls back to forward rendering (no deferred or light prepass support) on platforms which only support four render targets, such as many mobile GPUs.

See documentation on Rendering paths to learn more about forward and deferred rendering.

Advanced use

Mixed Lights can change their Transform and visual properties (such as colour or intensity) during run time, but only within strong limitations. In fact, because some lighting is baked (and therefore precomputed), changing any parameters at run time leads to inconsistent results when combining real-time and precomputed lighting.

In the case of **Baked Indirect** and **Shadowmask**, the direct lighting contribution behaves just like a **Realtime** Light, so you can change parameters like the color, intensity and even the Transform of the Light. However, baked values are precomputed, and cannot change at run time.

For example: if you bake a red **Mixed** Light into the light map, but change its color from red to green at run time, all direct lighting switches to the green color. However, all indirect lighting is baked into the light maps, so it remains red. The same applies to moving a Mixed Light at run time - direct lighting will follow the Light, but indirect lighting will remain at the position at which the Light was baked.

If only subtle changes are introduced to direct lighting (for example, by only slightly modifying the hue or intensity of a Light), it is possible to get the benefits of indirect lighting and for the Light to appear somewhat dynamic, without the extra processing time required for a **Realtime** Light. Indirect lighting is still incorrect, but the error might be subtle enough not to be objectionable. This works especially well for Lights without precomputed shadow information. This is achieved either by having shadows disabled for the Light, or by using Baked Indirect mode where shadows are real time. As shadowmasks are part of the direct lighting computation, moving such Lights causes visual inconsistencies with shadows not lining up correctly.

The following video shows an example of what happens when a Mixed Light is moved too far away from the spot where it was baked. Note how the indirect red light on the walls remains in place despite the object moving far away: https://youtu.be/o6pVBqrj8-s

The following video shows an example of how to slightly modify a Mixed Light without causing noticeable inconsistencies with indirect lighting: https://youtu.be/XN6ya31gm1I

Technical Details

In the case of Mixed Lights, the last segment of a light path (that is, the path from the Light to the surface) becomes part of the precomputation as well. However, Unity still handles direct lighting and indirect lighting separately. It bakes indirect lighting into light maps and **Light Probes**, which are then sampled at run time. Indirect lighting is generally low frequency, meaning it looks smooth and doesn't contain detailed shadows or light transitions. Therefore, shadows are handled with direct lighting where they have a high visible impact.

The difference in how shadows are precomputed and stored is reflected in the various submodes for Mixed Lights:

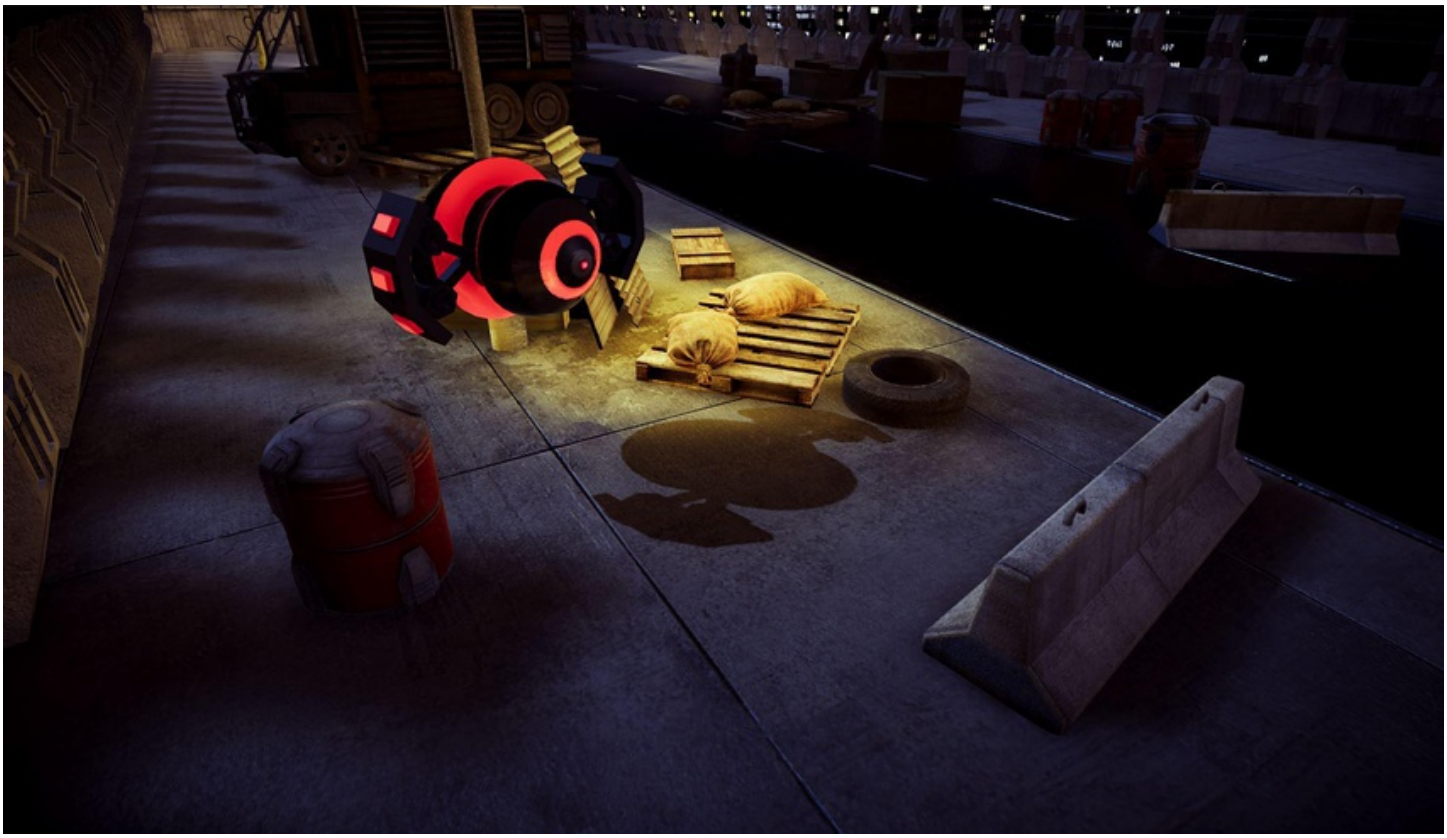
Baked Indirect
Shadowmask
Subtractive

Shadow information can be precomputed and stored in a shadowmask. A shadowmask is a Texture which shares the same UV layout and resolution with its corresponding light map. It stores occlusion information for up to four Lights per texel (because Textures are limited to up to four channels on current GPUs). The values range from 0 to 1, with values in-between marking soft shadow areas.

If shadowmasks are enabled, Light Probes also store occlusion information for up to four Lights. If more than four Lights intersect, the excess Lights fall back to **Baked Lights**. You can inspect this behavior with the shadowmask overlap visualization mode. This information is precomputed, so the only shadows Unity stores in the shadowmask are shadows cast from static GameObjects onto other static GameObjects. These shadows can have smoother edges providing better quality than real-time shadow maps, depending on the light map resolution. Because each Mixed Light retains its shadowmask channel mapping at run time, shadows cast by dynamic GameObjects via shadow maps can be correctly composited with precomputed shadows from static GameObjects, avoiding inconsistencies like double shadowing.

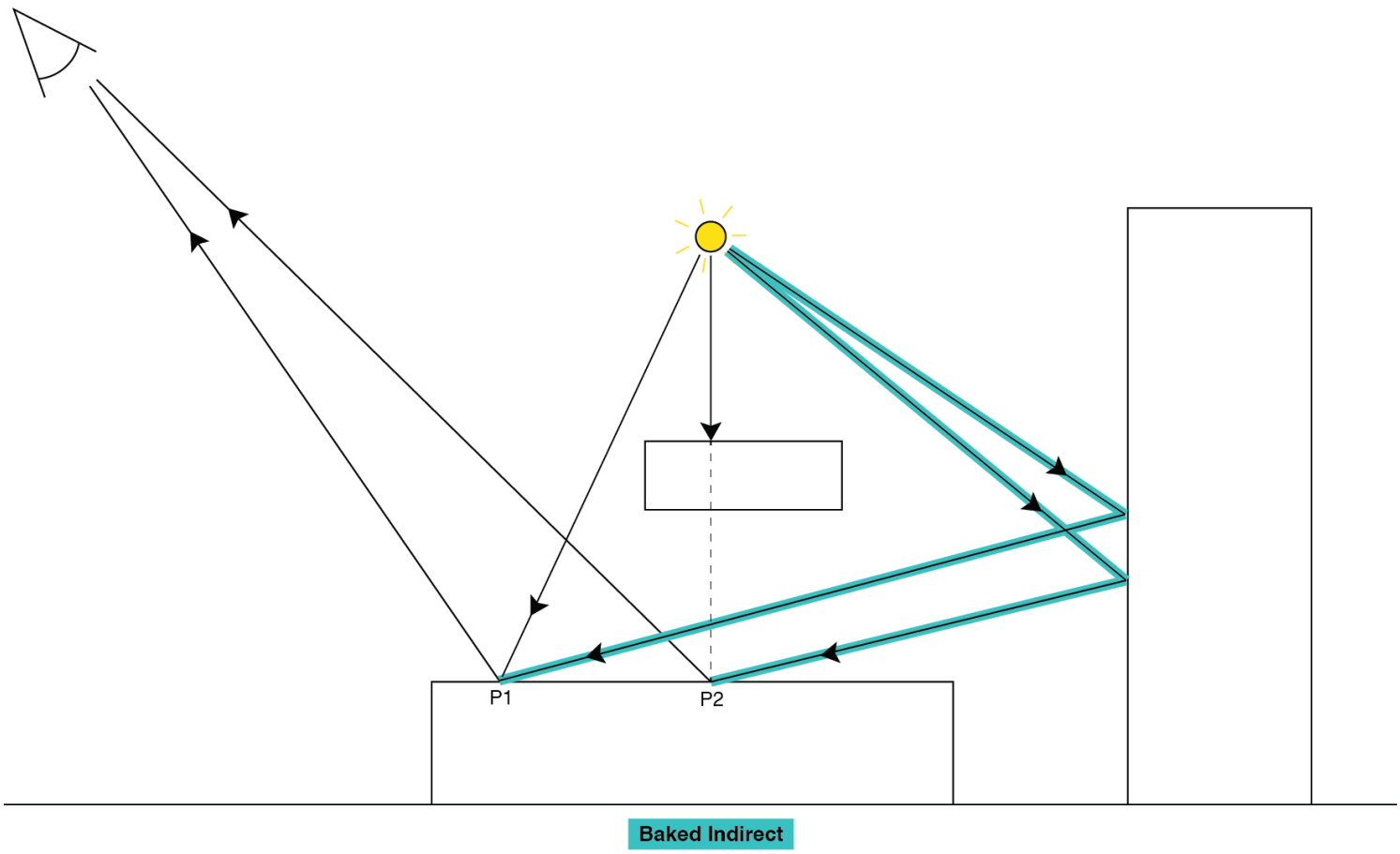
The only perceptible differences between shadows from static GameObjects and shadows from dynamic GameObjects are the differences in resolution and filtering of the precomputed shadowmask and the run-time shadow map, and that precomputed shadows support various forms of area Lights, so soft shadows can have more realistic penumbras.

In this image, the Baked shadowmask resolution is similar to the Realtime shadow resolution

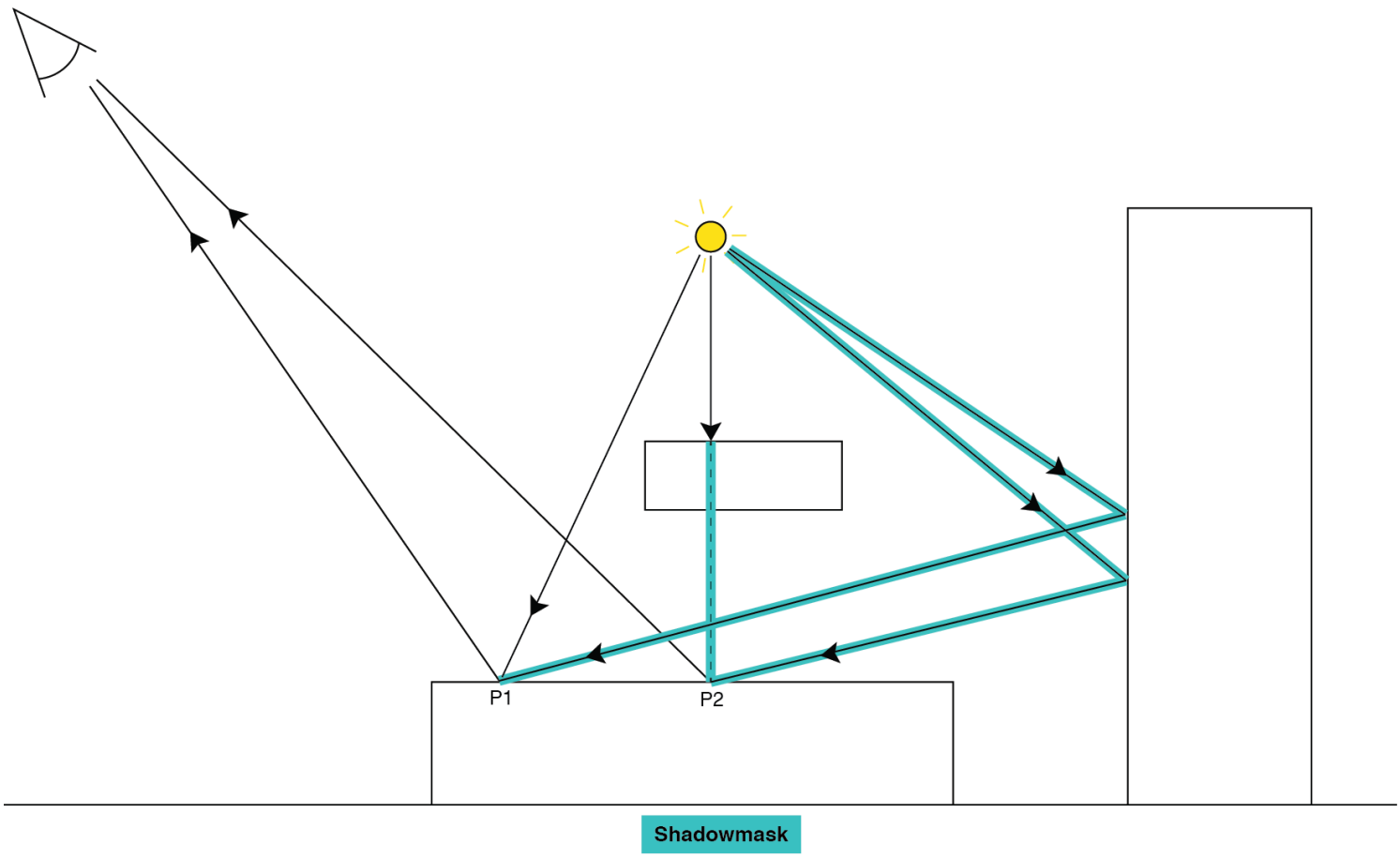


In this image, the Baked shadowmask resolution is much lower than the Realtime shadow resolution.

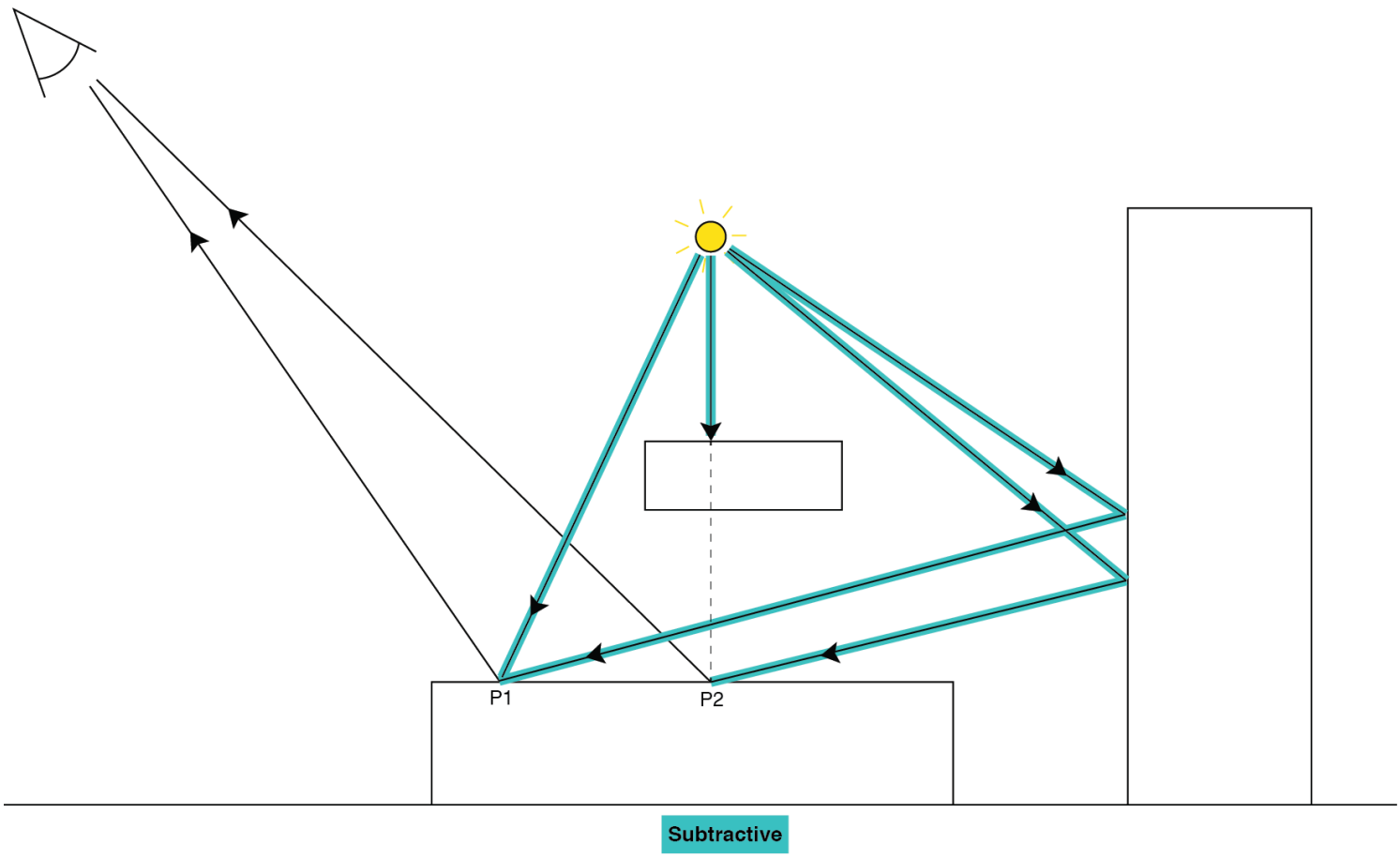
What **Baked Indirect** and **Shadowmask** have in common is that direct lighting is always computed in real time and added to the indirect lighting stored in the light map, so all Material effects that require a light direction continue to work. Dynamic GameObjects always cast shadows on other dynamic GameObjects via shadow maps within the **Shadow Distance** (**Edit** > **Project Settings** > **Quality** > **Shadows**), if shadows are enabled for that Light.



Baked Indirect mode: Only indirect lighting is precomputed



Shadowmask and **Distance Shadowmask** modes: Indirect lighting and direct occlusion are precomputed



Subtractive mode: All light paths are precomputed

2017–09–18 Page amended with limited editorial review

Light Modes added in 5.6

Baked Indirect mode

Leave feedback

Baked Indirect mode is a lighting mode shared by all **Mixed Lights** in a Scene. To set Mixed lighting to **Baked Indirect**, open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**), click the **Scene** tab, navigate to **Mixed Lighting** and set the **Lighting Mode** to **Baked Indirect**. See documentation on Mixed lighting to learn more about this lighting mode, and see documentation on Light modes to learn more about the other modes available.

For Lights that are set to **Baked Indirect** mode, Unity only precomputes indirect lighting, and does not carry out shadow pre-computations. Shadows are fully real-time within the **Shadow Distance** (menu: **Edit** > **Project Settings** > **Quality** > **Shadows**). In other words, **Baked Indirect** Lights behave like Realtime Lights with additional indirect lighting, but with no shadows beyond the **Shadow Distance**. You can use effects like the Post-processing fog effect to obscure the missing shadows past that distance.

A good example of when **Baked Indirect** mode might be useful is if you are building an indoor shooter or adventure game set in rooms connected with corridors. The viewing distance is limited, so everything that is visible should usually fit within the **Shadow Distance**. This mode is also useful for building a foggy outdoor Scene, because you can use the fog to hide the missing shadows in the distance.

Shadows

The following table shows how static and dynamic **GameObjects** cast and receive shadows when using **Baked Indirect** mode:

	Dynamic receiver A dynamic GameObject that is receiving a shadow from another static or dynamic GameObject		**Static receiver** A static GameObject that is receiving a shadow from another static or dynamic GameObject	
	Within Shadow Distance	Beyond Shadow Distance	Within Shadow Distance	Beyond Shadow Distance
Dynamic caster A dynamic GameObject that is casting a shadow	Shadow map	-	Shadow map	-
Static caster A static GameObject that is casting a shadow	Shadow map	-	Shadow map	-

Advantages and disadvantages of Baked Indirect mode

The performance requirements of **Baked Indirect** mode make it a good option for building to mid-range PCs and high-end mobile devices. These are the most significant advantages and disadvantages of using **Baked Indirect** mode:

Advantages

It provides the same visual effect as Realtime Lighting.
It provides real-time shadows for all combinations of static and dynamic GameObjects.
It provides indirect lighting.

Disadvantages

It has higher performance requirements relative to other **Mixed** Light modes, because it renders shadow-casting static GameObjects into shadow maps.
Shadows do not render beyond the **Shadow Distance**.

2017–06–08 Page published with limited editorial review

Light Modes added in 5.6

Shadowmask mode

Leave feedback

Shadowmask mode is a lighting mode shared by all Mixed Lights in a Scene. To set Mixed lighting to Shadowmask, open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**), click the **Scene tab**, navigate to **Mixed Lighting** and set the **Lighting Mode** to **Shadowmask**.

You also need to select the desired Shadowmask mode to use from the **Quality Settings** (menu: **Edit** > **Project Settings** > **Quality**):

Shadowmask: Static **GameObjects** that cast shadows always use baked shadows.
Distance Shadowmask: Unity uses real-time shadows up to the **Shadow Distance**, and baked shadows beyond.
See documentation on Mixed lighting and Light modes to learn more about the other modes available.

2017–09–18 Page amended with limited editorial review

Shadowmask modes added to Quality Settings in 2017.1

Shadowmask

Leave feedback

Shadowmask is a version of the **Shadowmask** lighting mode shared by all Mixed Lights in a Scene. To set Mixed lighting to **Shadowmask**:

Open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**), click the **Scene** tab, navigate to **Mixed Lighting** and set the **Lighting Mode** to **Shadowmask**.
Next, open the **Quality Settings** (menu: **Edit** > **Project Settings** > **Quality**), navigate to **Shadowmask Mode** and set it to **Shadowmask**.
See documentation on Mixed lighting to learn more about this lighting mode, and see documentation on Light modes to learn more about the other modes available.

A shadow mask is a Texture that shares the same UV layout and resolution with its corresponding **lightmap**. It stores occlusion information for up to 4 lights per texel, because Textures are limited to up to 4 channels on current GPUs.

Unity precomputes shadows cast from static **GameObjects** onto other static **GameObjects**, and stores them in a separate Shadowmask Texture for up to 4 overlapping lights. If more than 4 lights overlap, any additional lights fall back to Baked Lighting. Which of the lights fall back to Baked Lighting is determined by the baking system and stays consistent across bakes, unless one of the lights overlapping is modified. Light Probes also receive the same information for up to 4 lights.

Light overlapping is computed independently of shadow receiving objects. So, an object can get the influence of 10 different mixed lights all from the same Shadowmask/Probe channel, as long as those light **bounding volumes** don't overlap at any point in space. If some lights overlap then more channels are used. And, if a light does overlap while all 4 channels have already been assigned, that light will fallback to fully baked.

In **Shadowmask** mode:

Static GameObjects receive shadows from other static GameObjects via the shadow mask, regardless of the **Shadow Distance** (menu: **Edit** > **Project Settings** > **Quality** > **Shadows**). They also receive shadows from dynamic GameObjects, but only those within the **Shadow Distance**.

Dynamic GameObjects receive shadows from other dynamic GameObjects within the **Shadow Distance** via shadow maps. They also receive shadows from static GameObjects, via Light Probes. The shadow fidelity depends on the density of Light Probes in the Scene, and the **Light Probes** mode selected on the **Mesh** Renderer.

Unity automatically composites overlapping shadows from static and dynamic GameObjects, because shadow masks (which hold static GameObject lighting and shadow information) and shadow maps (which hold dynamic GameObject lighting and shadow information) only encode occlusion information.

A good example of when **Shadowmask** mode might be useful is if you are building an almost fully static Scene, using specular Materials, soft baked shadows and a dynamic shadow caster, not too close to the **camera**. Another good example is an open-world Scene with baked shadows up to the horizon, but without dynamic lighting such as a day/night cycle.

Shadows

The following table shows how static and dynamic GameObjects cast and receive shadows when using **Shadowmask** mode:

	Dynamic receiver A dynamic GameObject that is receiving a shadow from another static or dynamic GameObject		**Static receiver** A static GameObject that is receiving a shadow from another static or dynamic GameObject	
	Within Shadow Distance	Beyond Shadow Distance	Within Shadow Distance	Beyond Shadow Distance

	Dynamic receiver A dynamic GameObject that is receiving a shadow from another static or dynamic GameObject		**Static receiver** A static GameObject that is receiving a shadow from another static or dynamic GameObject	
Dynamic caster A dynamic GameObject that is casting a shadow	Shadow map	-	Shadow map	-
Static caster A static GameObject that is casting a shadow	Light Probes	Light Probes	Shadowmask	Shadowmask

Advantages and disadvantages of Shadowmask mode

The performance requirements of **Shadowmask** mode make it a good option for building to low and mid-range PCs, and mobile devices. These are the most significant advantages and disadvantages of using **Shadowmask** mode:

Advantages

It offers the same visual effect as Realtime Lighting.

It provides real-time shadows from dynamic GameObjects onto static GameObjects.

One Texture operation in the **Shader** handles all lighting and shadows between static GameObjects.

It automatically composites overlapping shadows from static and dynamic GameObjects.

It has mid-to-low performance requirements, because it does not render static GameObjects into shadow maps.

It provides indirect lighting.

Disadvantages

It only provides low-resolution shadows from static GameObjects onto dynamic GameObjects, via Light Probes.

It only allows up to 4 overlapping light volumes (see documentation under the 'Technical details' section of Mixed Lighting for more information).

It has increased memory requirements for the light map Texture set.

It has increased memory requirements for the shadow mask Texture.

2017–09–18 Page amended with limited editorial review

Light Modes added in 5.6

Added advice on light overlap computation in Unity 2017.1

Distance Shadowmask

Leave feedback

Distance Shadowmask is a version of the **Shadowmask** lighting mode. It is shared by all Mixed Lights in a Scene. To set **Mixed lighting** to **Distance Shadowmask**:

Open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**), click the **Scene** tab, navigate to **Mixed Lighting**, and set the **Lighting Mode** to **Shadowmask**.
Next, open the **Quality Settings** (menu: **Edit** > **Project Settings** > **Quality**), navigate to **Shadowmask Mode** and set it to **Distance Shadowmask**.
See documentation on Mixed lighting to learn more about this lighting mode, and see documentation on Light modes to learn more about the other modes available.

A shadow mask is a Texture that shares the same UV layout and resolution with its corresponding light map. It stores occlusion information for up to 4 lights per texel, because Textures are limited to up to 4 channels on current GPUs.

The **Distance Shadowmask** mode is a version of the **Shadowmask** lighting mode that includes high quality shadows cast from static **GameObjects** onto dynamic **GameObjects**.

Within the **Shadow Distance** (menu: **Edit** > **Project Settings** > **Quality** > **Shadows**), Unity renders both dynamic and static GameObjects into the shadow map, allowing static GameObjects to cast sharp shadows onto dynamic GameObjects. For this reason, **Distance Shadowmask** mode has higher performance requirements than **Shadowmask** mode, because all static GameObjects in the scene are rendered into the shadow map.

Beyond the **Shadow Distance**:

Static GameObjects receive high-resolution shadows from other static GameObjects via the precomputed shadow mask.
Dynamic GameObjects receive low-resolution shadows from static GameObjects via Light Probes.
A good example of when **Distance Shadowmask** mode might be useful is if you are building an open world Scene with shadows up to the horizon, and complex static Meshes casting real-time shadows on moving characters.

Shadows

The following table shows how static and dynamic GameObjects cast and receive shadows when using **Distance Shadowmask** mode:

	Dynamic receiver A dynamic GameObject that is receiving a shadow from another static or dynamic GameObject		**Static receiver** A static GameObject that is receiving a shadow from another static or dynamic GameObject	
	Within Shadow Distance	Beyond Shadow Distance	Within Shadow Distance	Beyond Shadow Distance
Dynamic caster A dynamic GameObject that is casting a shadow	Shadow map	-	Shadow map	-
Static caster A static GameObject that is casting a shadow	Shadow map	Light Probes	Shadow map	Shadow mask

Advantages and disadvantages of Distance Shadowmask mode

The performance requirements of **Distance Shadowmask** mode make it a good option for building to high-end PCs and current generation consoles. These are the most significant advantages and disadvantages of using **Distance Shadowmask** mode:

Advantages

It provides the same visual effect as Realtime Lighting.
It provides real-time shadows from dynamic GameObjects onto static GameObjects, and static GameObjects onto dynamic GameObjects.
One Texture operation in the **Shader** handles all lighting and shadows between static GameObjects.
It automatically composites dynamic and static shadows.
It provides indirect lighting.

Disadvantages

It only allows up to 4 overlapping light volumes (see documentation under the 'Technical details' section of Mixed lighting for more information).
It has increased memory requirements for light map Texture sets.
It has increased memory requirements for shadow mask Textures.
It has increased performance requirements, because Unity renders light and shadow from static GameObjects into shadow maps.

2017–09–18 Page published with limited editorial review

Light Modes added in 5.6

Subtractive mode

Leave feedback

Subtractive mode is a lighting mode shared by all Mixed Lights in a Scene. To set Mixed lighting to **Subtractive**, open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**), click the **Scene** tab, navigate to **Mixed Lighting** and set the **Lighting Mode** to **Subtractive**. See documentation on Mixed lighting to learn more about this lighting mode, and see documentation on Light modes to learn more about the other modes available.

Subtractive is the only Mixed lighting mode that bakes direct lighting into the light map, and discards the information that Unity uses to composite dynamic and static shadows in other Mixed lighting modes. Because the Light is already baked into the **lightmap**, Unity cannot perform any direct lighting calculations at run time.

In **Subtractive** mode:

Static **GameObjects** do not show any specular or glossy highlights at all from Mixed Lights. They also cannot receive any shadows from dynamic **GameObjects**, except for the main Directional Light (see paragraph below for more information on this).

Dynamic GameObjects receive real-time lighting, and support glossy reflections. However, they can only receive shadows from static GameObjects via Light Probes.

In **Subtractive** mode, the main Directional Light (which is usually the sun) is the only light source which casts real-time shadows from dynamic GameObjects onto static GameObjects. Shadows cast from static GameObjects onto other static GameObjects are baked into the lightmap, even for the main Light, so Unity cannot guarantee correct composition of baked and real-time shadows. **Subtractive** mode therefore has a Realtime Shadow Color field. Unity uses this color in the **Shader** to composite real-time shadows with baked shadows. To do this, it reduces the effect of the light map in areas shadowed by dynamic GameObjects. Because there is no correct value that the engine can predetermine, choosing a value that works for any given Scene is down to your own artistic choice.

A good example of when **Subtractive** mode might be useful is when you are building a cel-shaded (that is, cartoon-style) game with outside levels and very few dynamic GameObjects.

Shadows

The following table shows how static and dynamic GameObjects cast and receive shadows when using **Subtractive** mode:

	Dynamic receiver A dynamic GameObject that is receiving a shadow from another static or dynamic GameObject		**Static receiver** A static GameObject that is receiving a shadow from another static or dynamic GameObject	
	Within Shadow Distance	Beyond Shadow Distance	Within Shadow Distance	Beyond Shadow Distance
Dynamic caster A dynamic GameObject that is casting a shadow	Shadow map	-	Main light shadow map	-

	Dynamic receiver A dynamic GameObject that is receiving a shadow from another static or dynamic GameObject		**Static receiver** A static GameObject that is receiving a shadow from another static or dynamic GameObject	
Static caster A static GameObject that is casting a shadow	Light Probes	Light Probes	Lightmap	Lightmap

Advantages and disadvantages of Subtractive mode

The performance requirements of **Subtractive** mode make it a good option for building to low-end mobile devices. These are the most significant advantages and disadvantages of using **Subtractive** mode:

Advantages

It provides high-quality shadows between static GameObjects in lightmaps at no additional performance requirement.

One Texture operation in the Shader handles all lighting and shadows between static GameObjects.

It provides indirect lighting.

Disadvantages

It does not provide real-time direct lighting, and therefore does not provide specular lighting.

It does not provide dynamic shadows on static GameObjects, except for one Directional Light (the main Light).

It only provides low-resolution shadows from static GameObjects onto dynamic GameObjects, via Light Probes.

It provides inaccurate composition of dynamic and static shadows.

It has increased memory requirements for the light map texture set (compared to no lightmaps).

2017–06–08 Page published with limited editorial review

Light Modes added in 5.6

Baked lighting

Leave feedback

Baked Lights are Light components which have their **Mode** property set to **Baked**.

Use **Baked** mode for Lights used for local ambience, rather than fully featured Lights. Unity pre-calculates the illumination from these Lights before run time, and does not include them in any run-time lighting calculations. This means that there is no run-time overhead for **baked Lights**.

Unity bakes direct and indirect lighting from baked Lights into light maps (to illuminate static GameObjects) and Light Probes (to illuminate dynamic Light GameObjects). Baked Lights cannot emit specular lighting, even on dynamic **GameObjects** (see Wikipedia: Specular highlight for more information). Baked Lights do not change in response to actions taken by the player, or events which take place in the **Scene**. They are mainly useful for increasing brightness in dark areas without needing to adjust all of the lighting within a **Scene**.

Baked Lights are also the only Light type for which dynamic GameObjects cannot cast shadows on other dynamic GameObjects.

Advantages of baked lighting

High-quality shadows from statics GameObjects on statics GameObjects in the light map at no additional cost.

Offers indirect lighting.

All lighting for static GameObjects can be just one Texture fetched from the light map in the **Shader**.

Disadvantages of baked lighting

No real-time direct lighting (that is, no specular lighting effects).

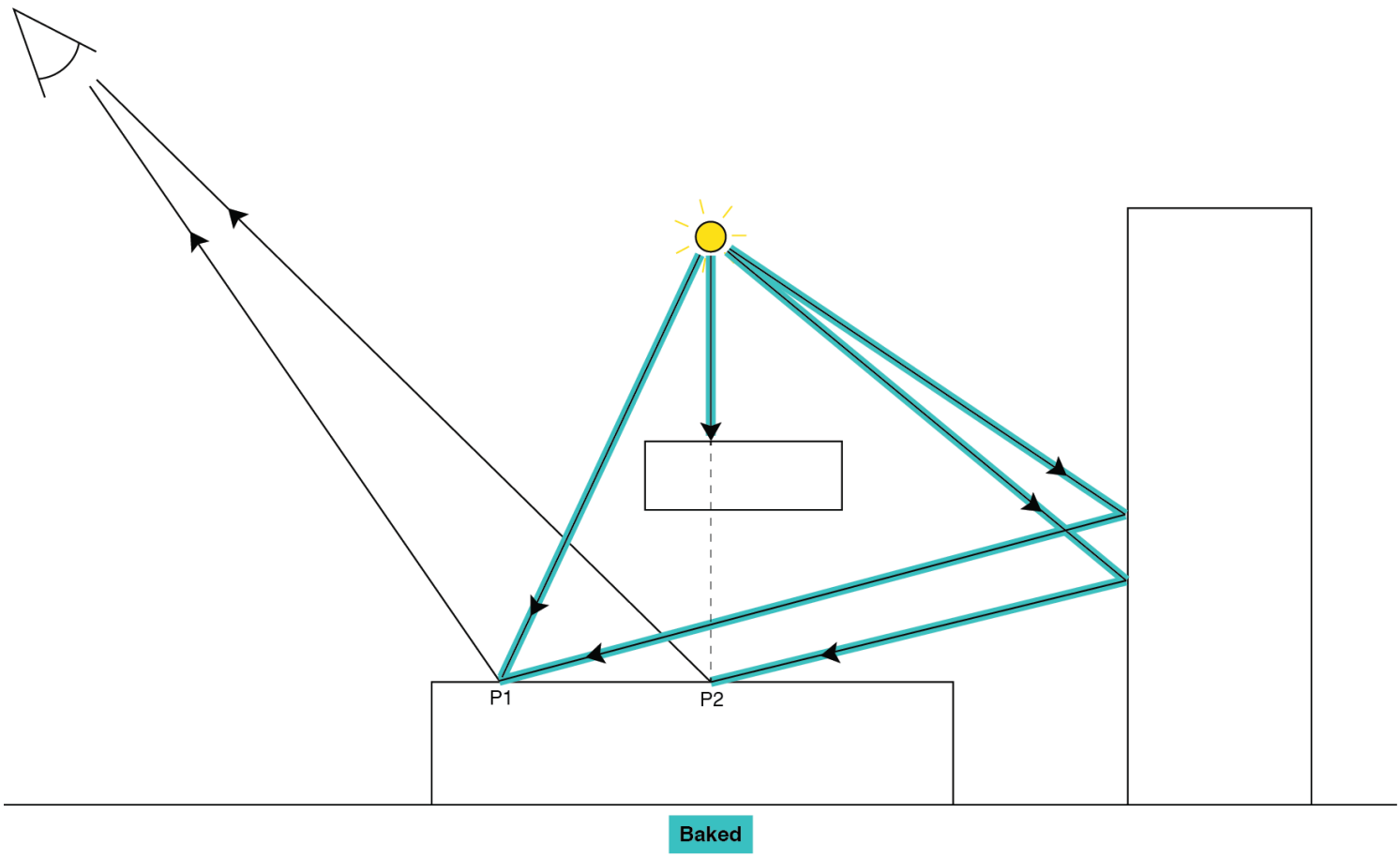
No shadows from dynamic GameObjects on static GameObjects.

You only get low-resolution shadows from static GameObjects on dynamic GameObjects using Light Probes.

Increased memory requirements compared to real-time lighting for the light map texture set, because light maps need to be more detailed to contain direct lighting information.

Technical details

For baked Lights, Unity precomputes the entire light path, except for the path segment from the **Camera** to the Surface. See documentation on Light Modes for more information about light paths.



Baked Mode: All light paths are precomputed

Unity also precomputes direct baked lighting, which means that light direction information is not available to Unity at run time. Instead, a small number of Texture operations handle all light calculations for baked Lights in the Scene area. Without this information, Unity cannot carry out calculations for specular and glossy reflections. If you need specular reflections, use Reflection Probes or use Mixed or Realtime lights. See documentation on directional light maps for more information.

Baked Lights never illuminate dynamic GameObjects at run time. The only way for dynamic GameObjects to receive light from baked Lights is via Light Probes. This is also the only difference between Baked Lights and any Subtractive mode Mixed Lights (except the main directional Light) which compute direct lighting on dynamic GameObjects at run time.

2017–06–08 Page published with limited editorial review

Light Modes added in 5.6

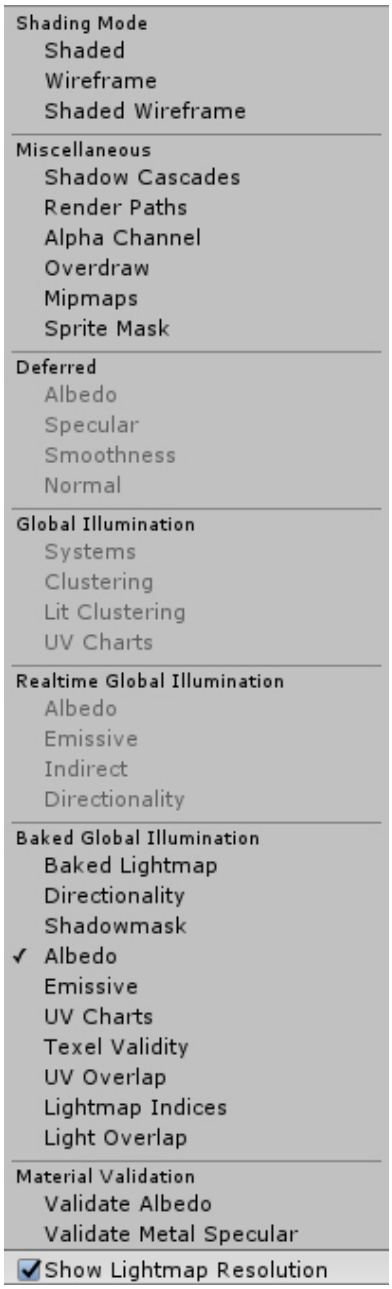
GI visualizations in the Scene view

Leave feedback

The **Scene view** has a number of Draw Modes to help you visualize different aspects of the **Scene**'s content. Among these are a set of modes to let you see exactly how global illumination (GI) is affecting your Scene. This page goes into more detail about the visualisation modes that are most relevant to GI.

Note that, in the Lighting window, the **Object** tab displays some of the different modes for the selected **GameObject**, with its UV channel rendered in Texture space as a wireframe **Mesh**. Tick the **Show** Lightmap__ Resolution__ checkbox to apply a checkerboard texture on top of each view, scaled to show the resolution.

GI visualizations are not designed to work in Play Mode.



Shading Mode

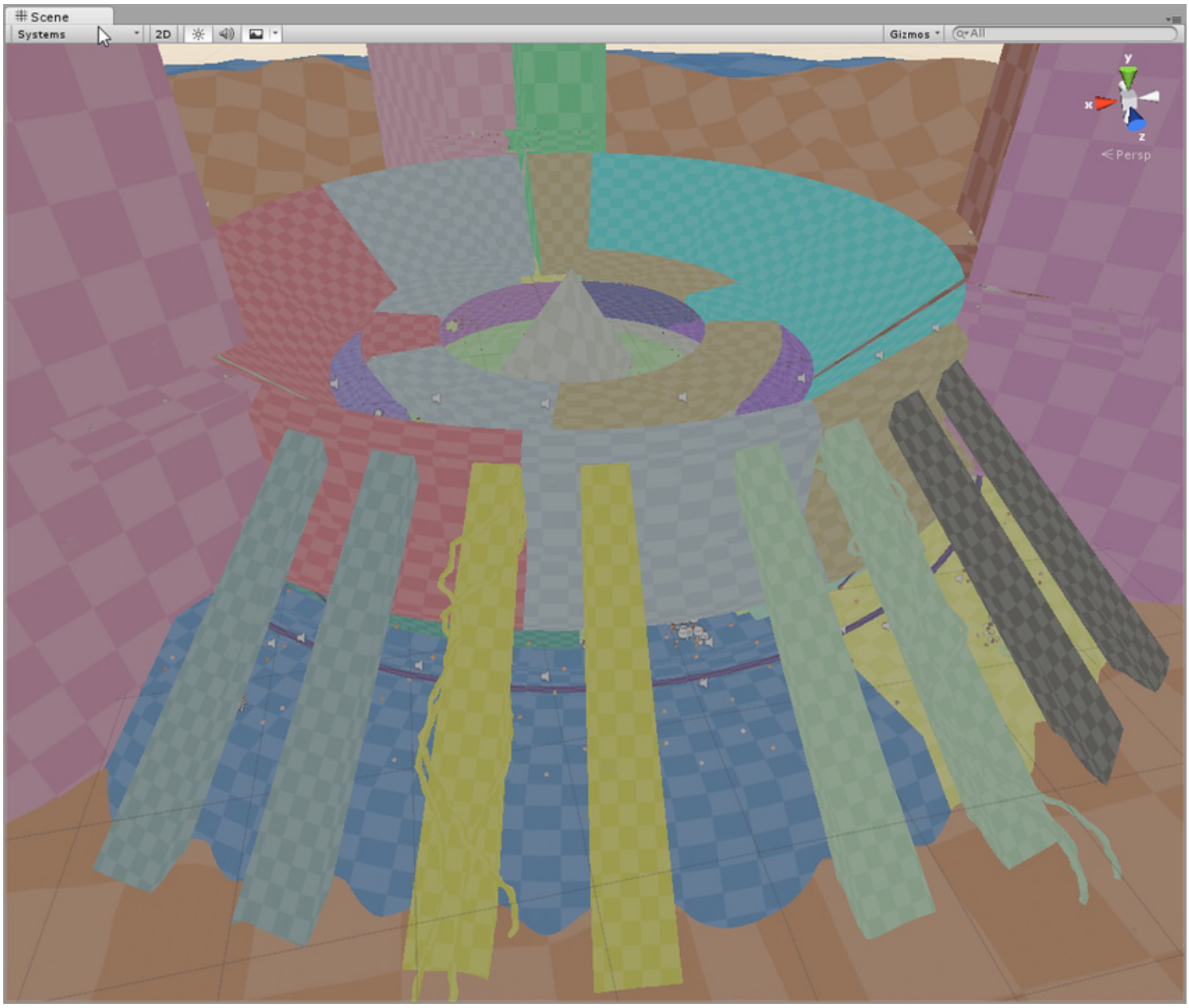
Shaded

The default **Shading Mode** is **Shaded**. This shows the Scene fully lit according to the current lighting setup.

Global Illumination

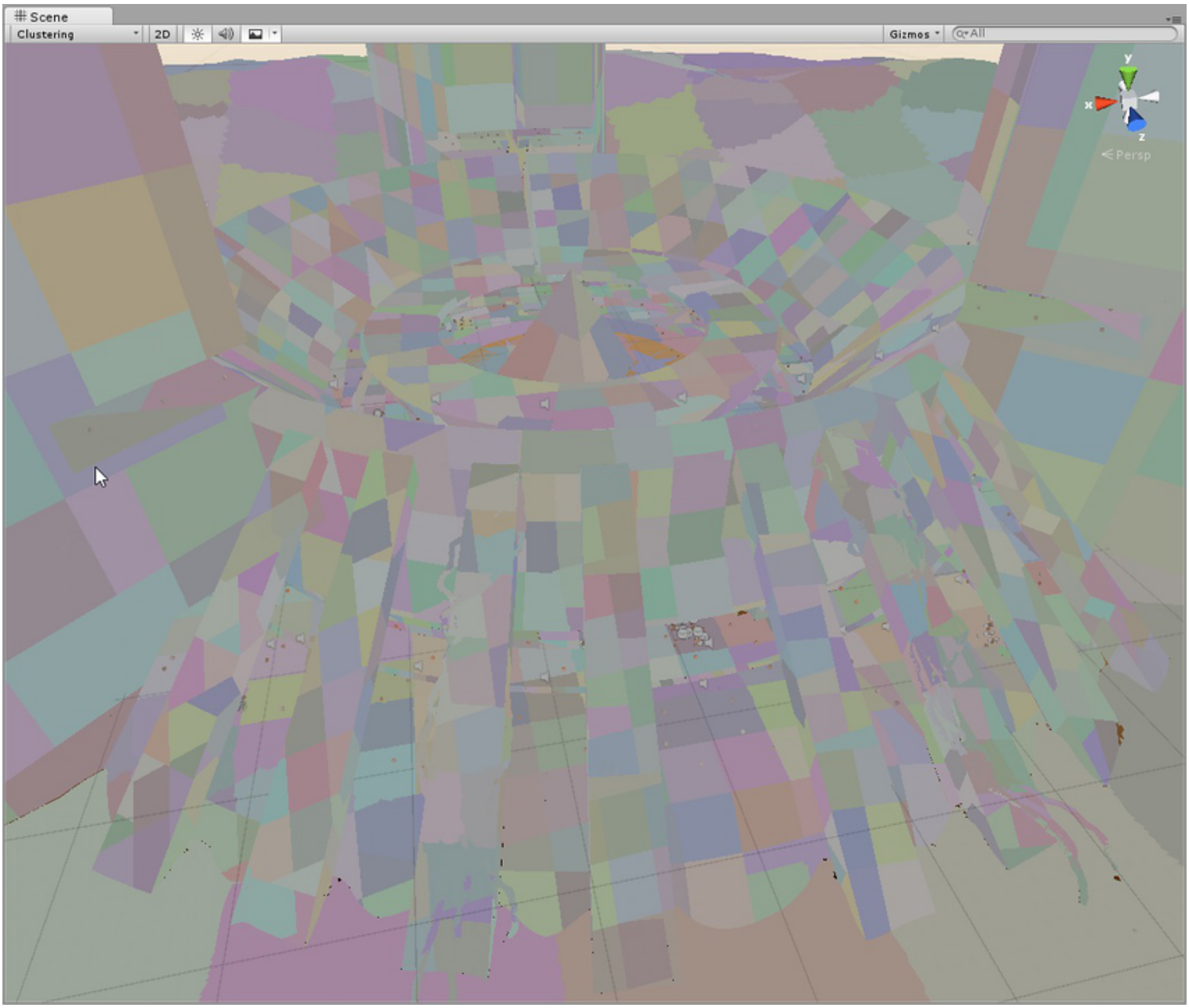
Systems

The precompute stage will automatically subdivide the scene into systems (i.e. groups of objects sharing the same realtime lightmap) based on proximity and Lightmap Parameters. This is done to allow multithreading and optimizations when updating indirect lighting. This visualization shows the systems with different colors.



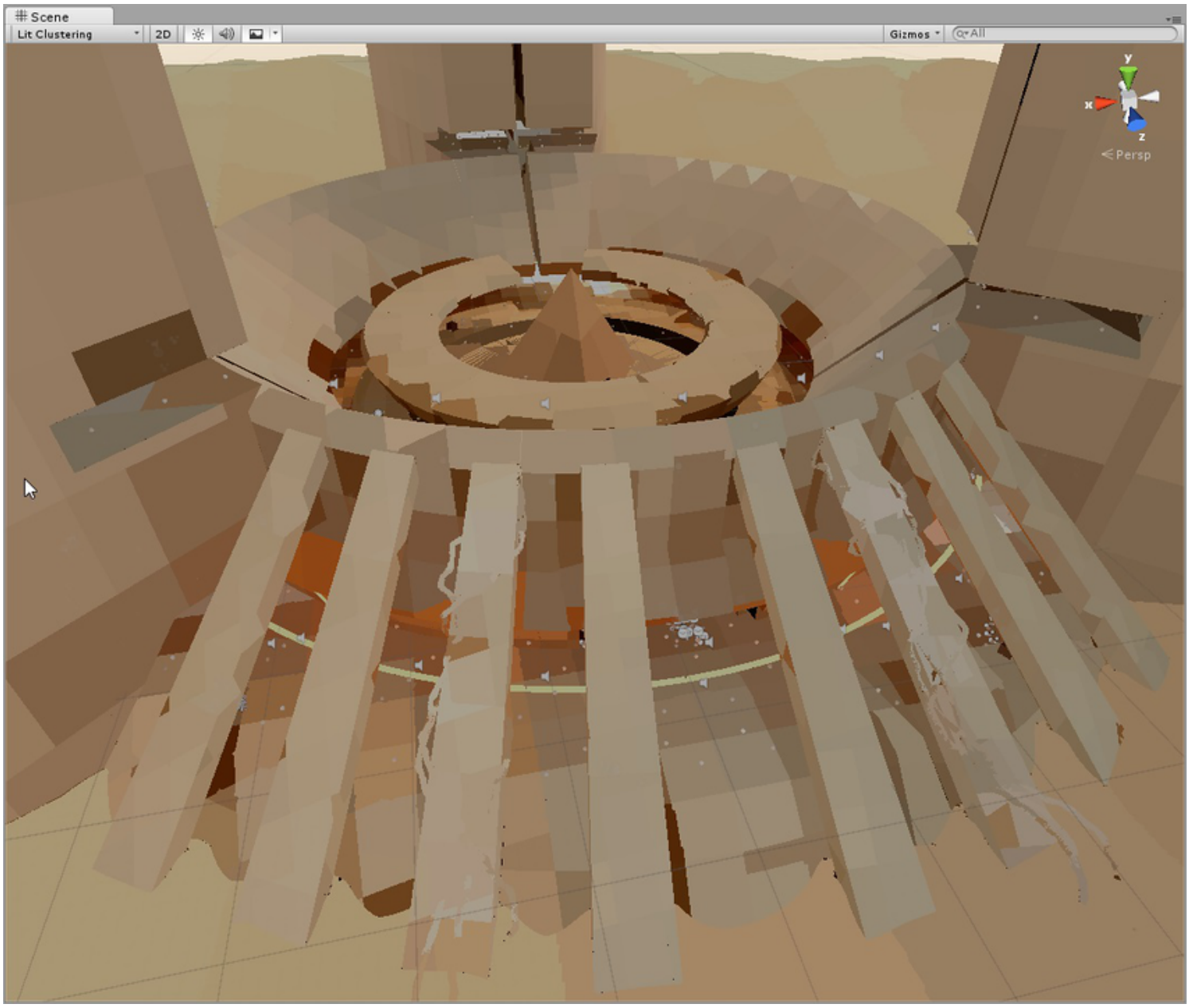
Clustering

This shows the clusters that **Enlighten** generates from the lightmap static geometry. **Enlighten** calculates indirect lighting using clusters that are generated in the Clustering step. Resulting clusters should be larger than lightmap texels (the ratio is controlled by the Cluster Resolution parameter in Lightmap Parameters. The step where geometry is converted to clusters can be quite memory intensive if the scale isn't set correctly. If you are seeing high memory usage or long baking times it could be because the static geometry in your scene is getting cut up into many more clusters than what is actually needed. The clustering scene view mode can help you identify the geometry that needs to have UVs or Realtime Resolution tweaked.



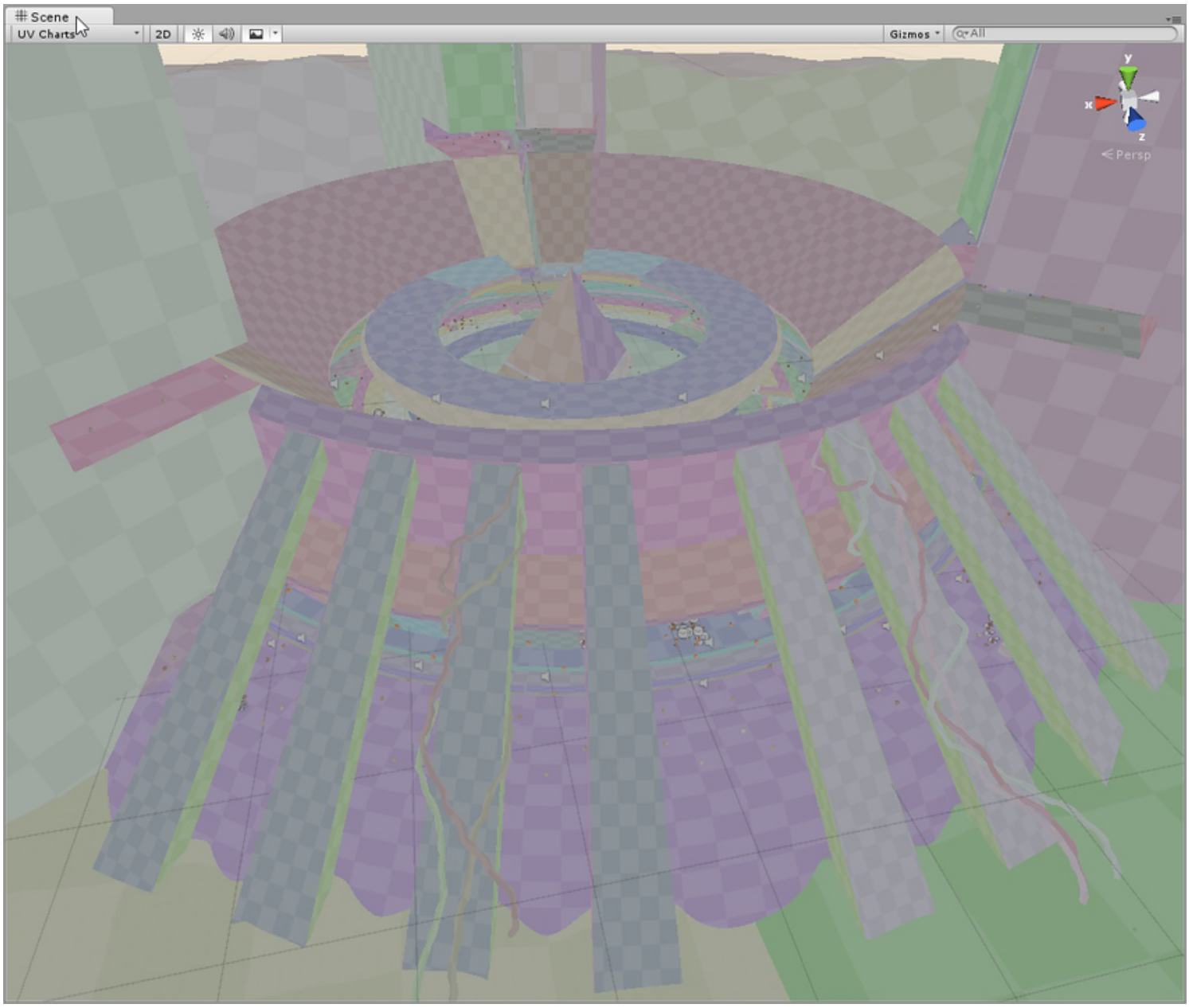
Lit Clustering

Those are the same clusters as seen in the Clustering view, but with realtime GI applied.



UV Charts

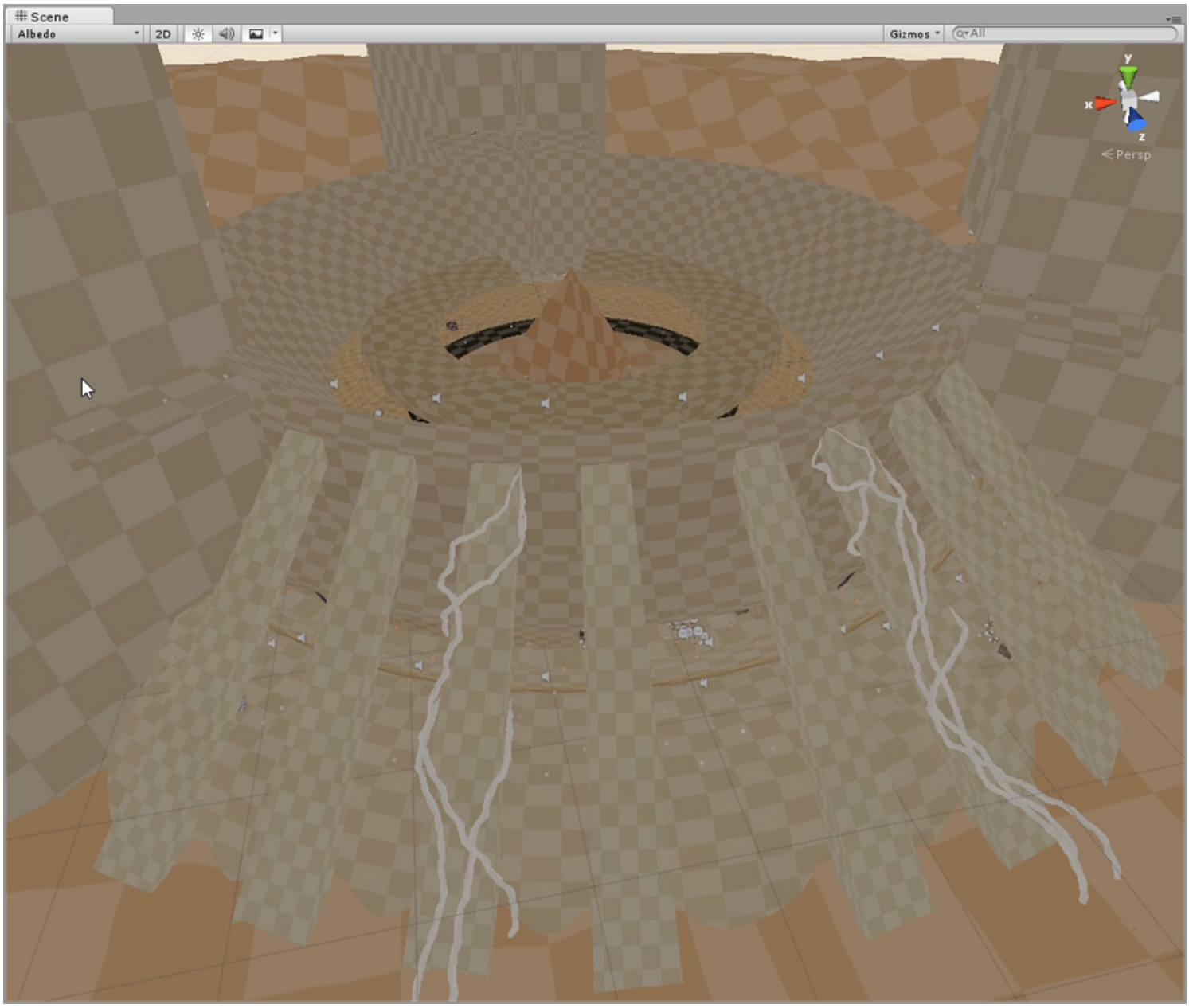
This shows the optimized UV layout used when calculating precomputed realtime GI. It is automatically generated during the precompute process. It is available as soon as the Instance precompute stage is completed. The UV Charts scene view mode can help you identify the geometry that needs to have UVs or scale adjusted (use the Resolution parameter in Lightmap Parameters to change scale). This view is also useful when adjusting the Realtime Resolution. Each chart has a different color.



Realtime Global Illumination

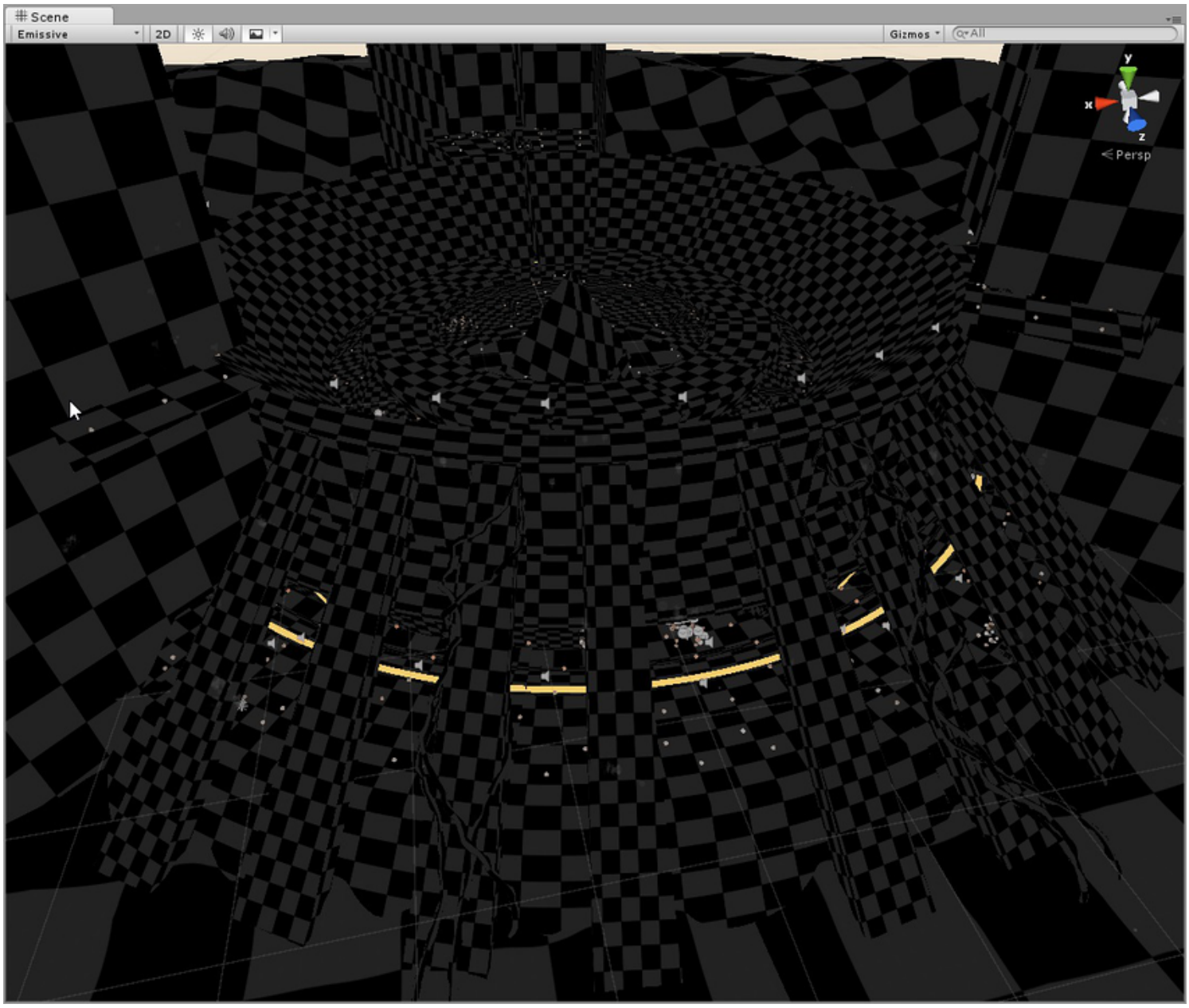
Albedo

This shows the albedo used when calculating GI. The albedo is generated from the material information and can be customized fully by adding a custom meta pass. The checkered overlay shows the resolution of the albedo texture that is passed to Enlighten.



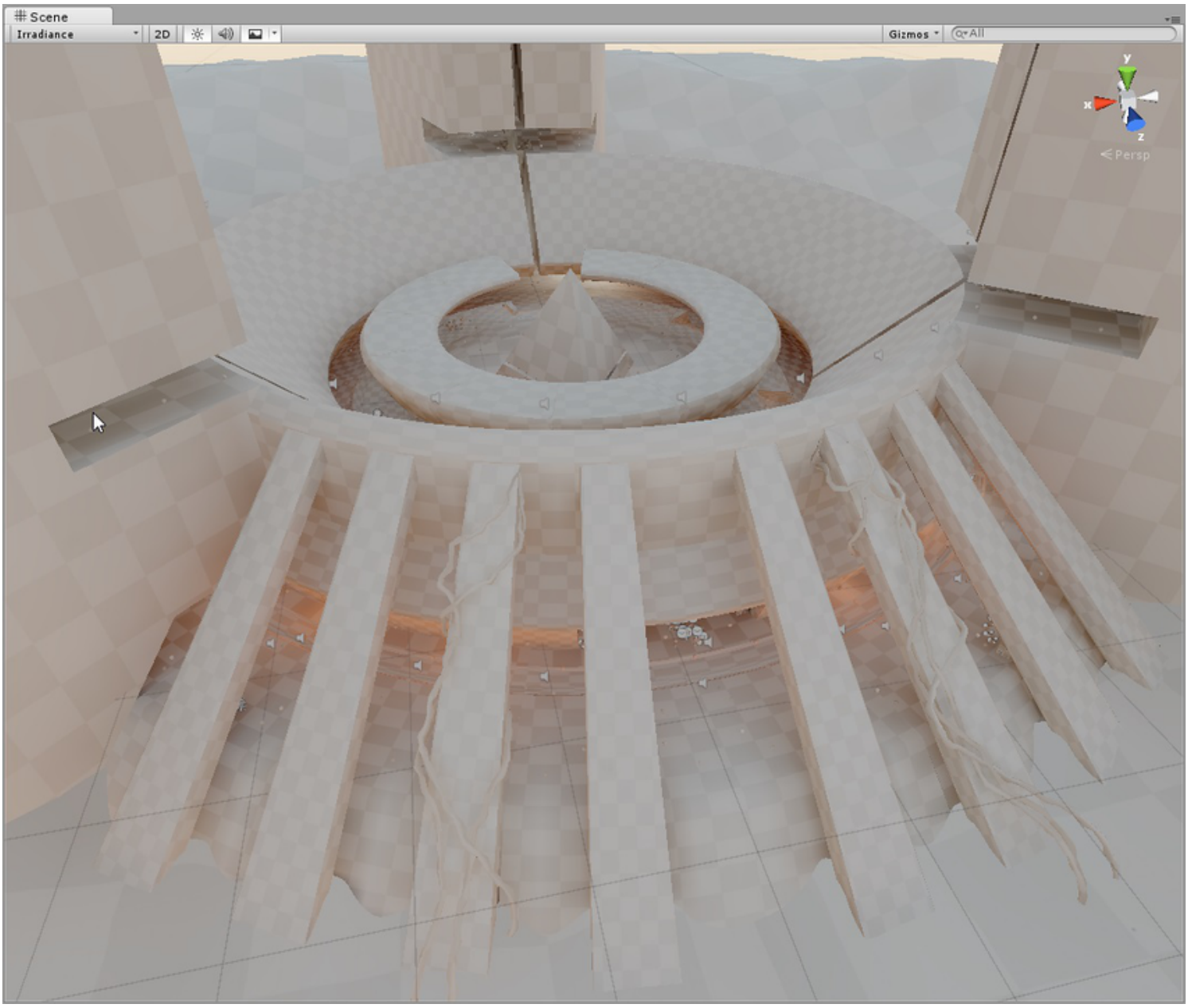
Emissive

Shows the emission used when calculating the GI. It is generated from the material information and can be fully customized by adding a custom meta pass. The checkered overlay shows the resolution of the emission texture that is passed into Enlighten.



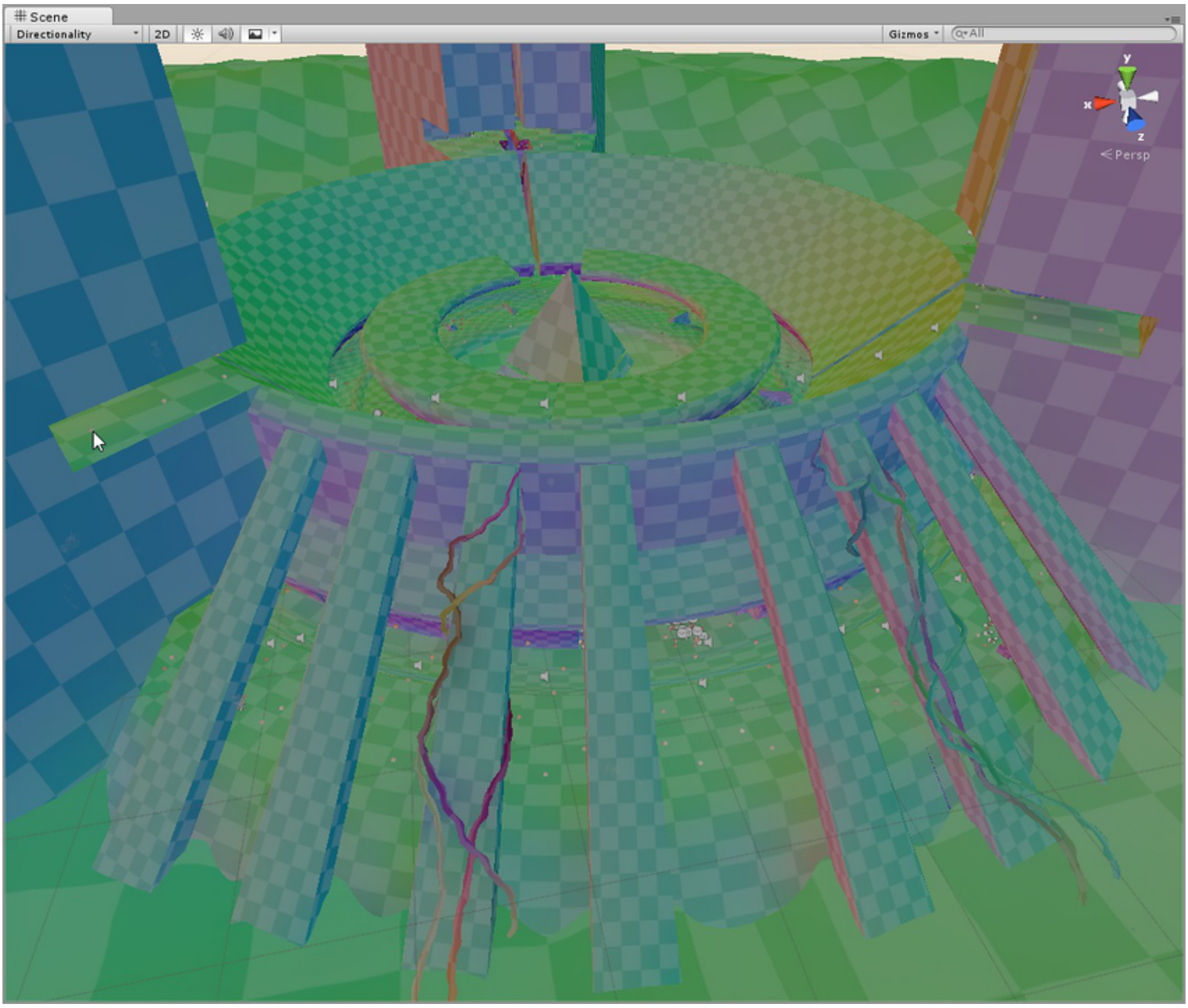
Indirect

This shows the indirect lighting only (the contents of realtime GI lightmaps generated by Enlighten). The checkered overlay shows the resolution of the irradiance texture. If Realtime GI is disabled, this view mode isn't selectable.



Directionality

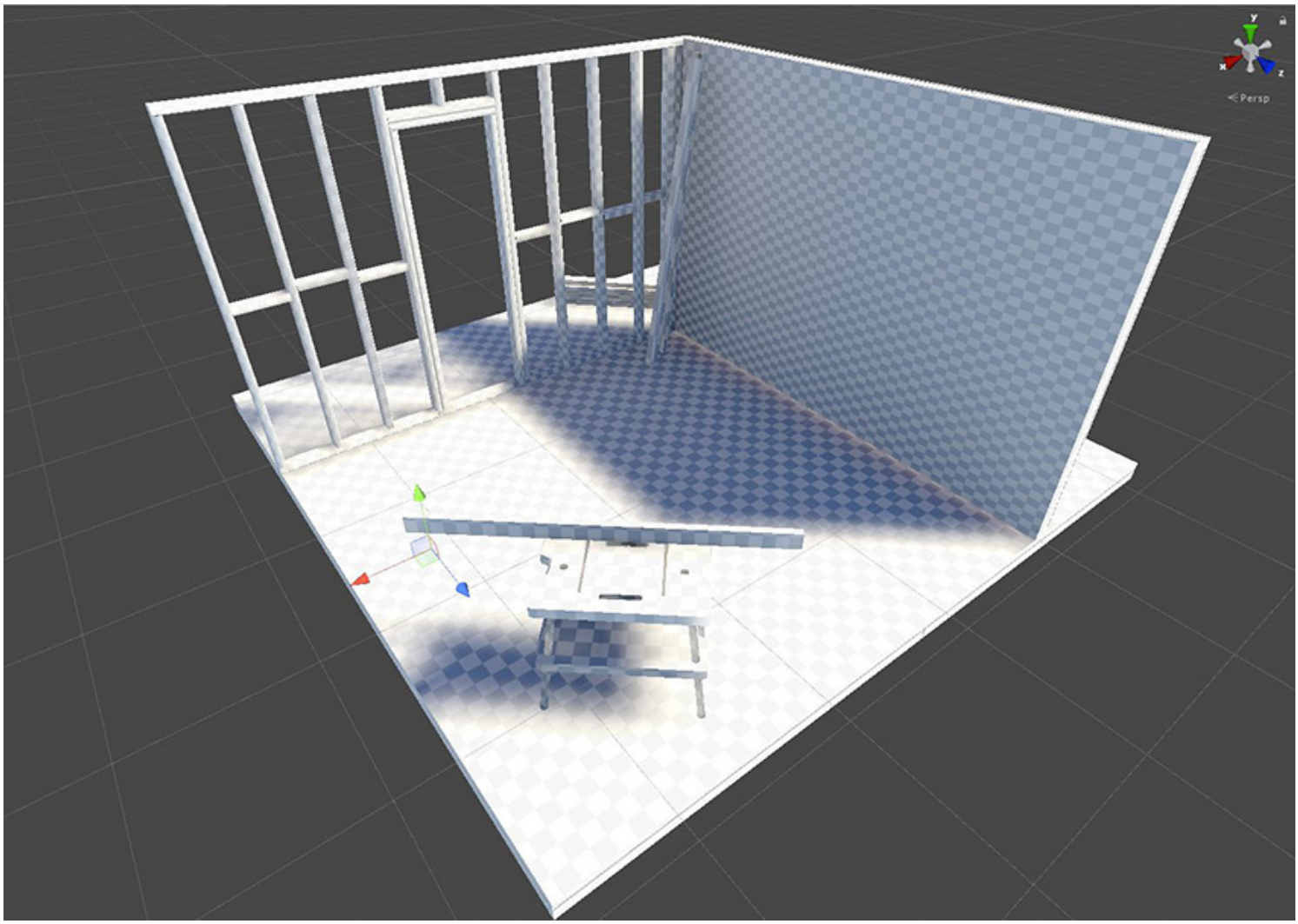
This view shows the most dominant light direction vector. Please refer to the Directional Lightmapping page for more info. The checkered overlay shows the resolution of the directionality texture.



Baked Global Illumination

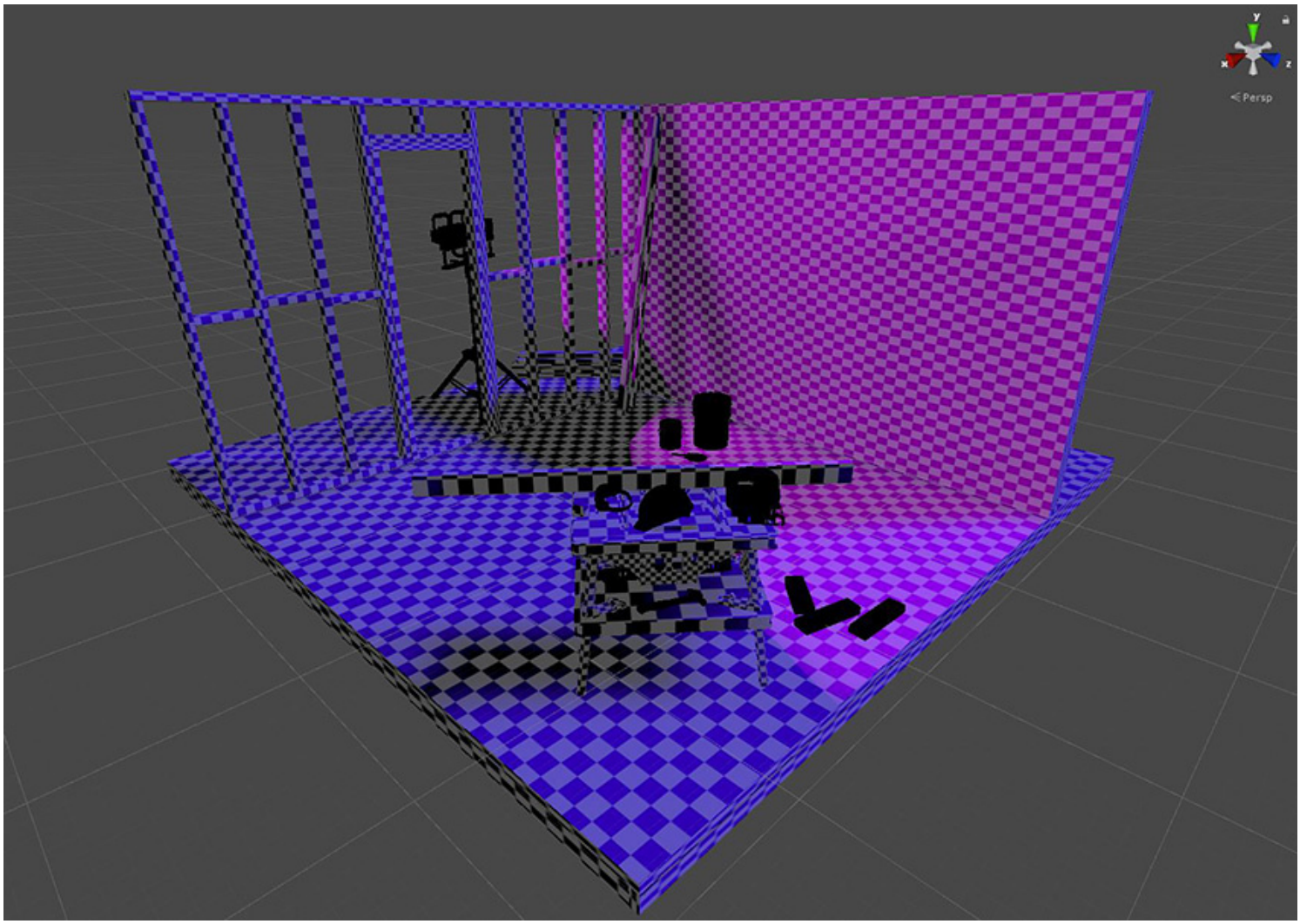
Baked Lightmap

This shows the baked lightmaps applied to the scene geometry. The checkered overlay shows the baked lightmap resolution.



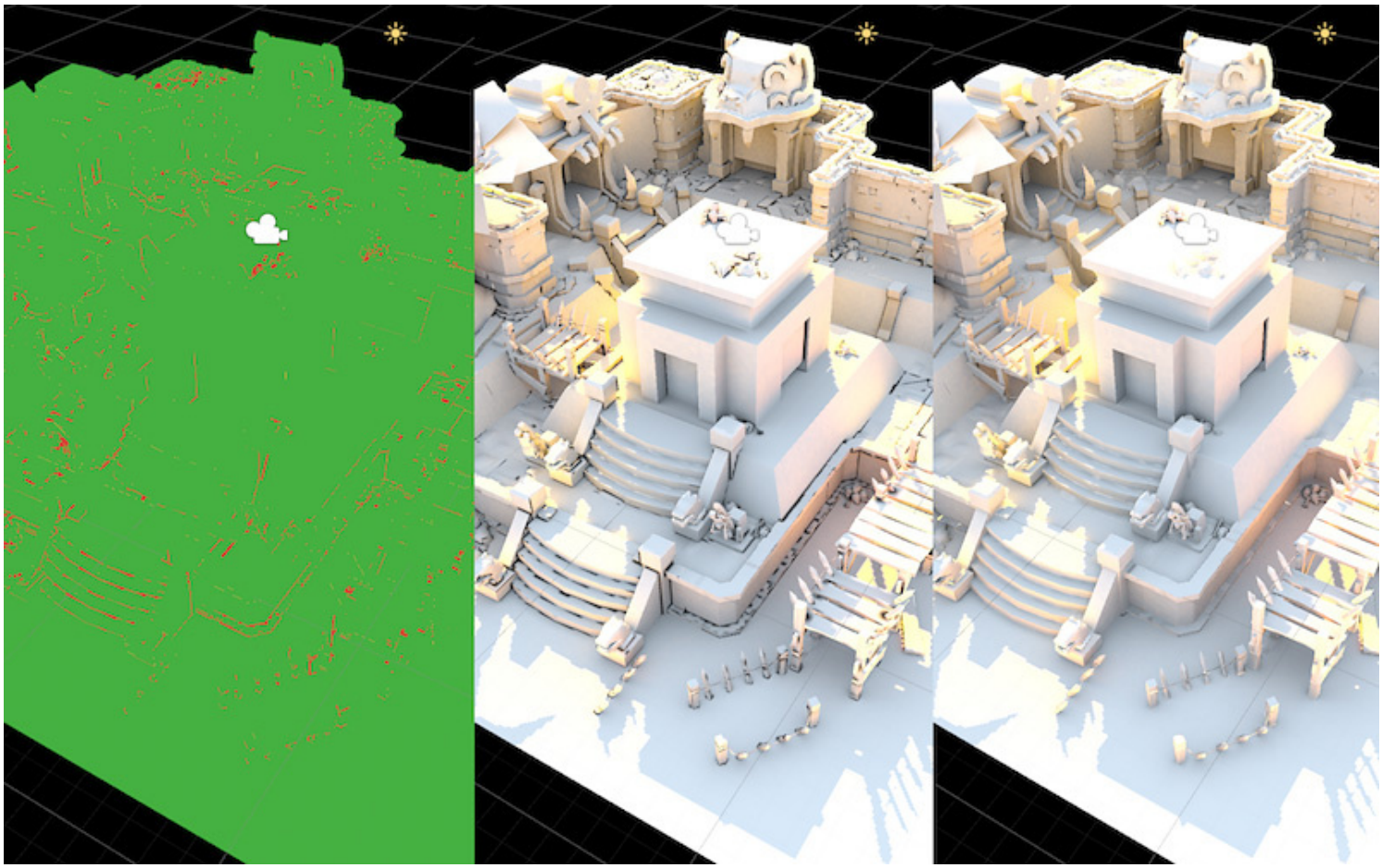
Shadowmask

This displays the **shadowmask** texture occlusion values. It colors the mesh and the light **gizmo** in the same color so one can verify that the light occlusion factors have been baked as expected.



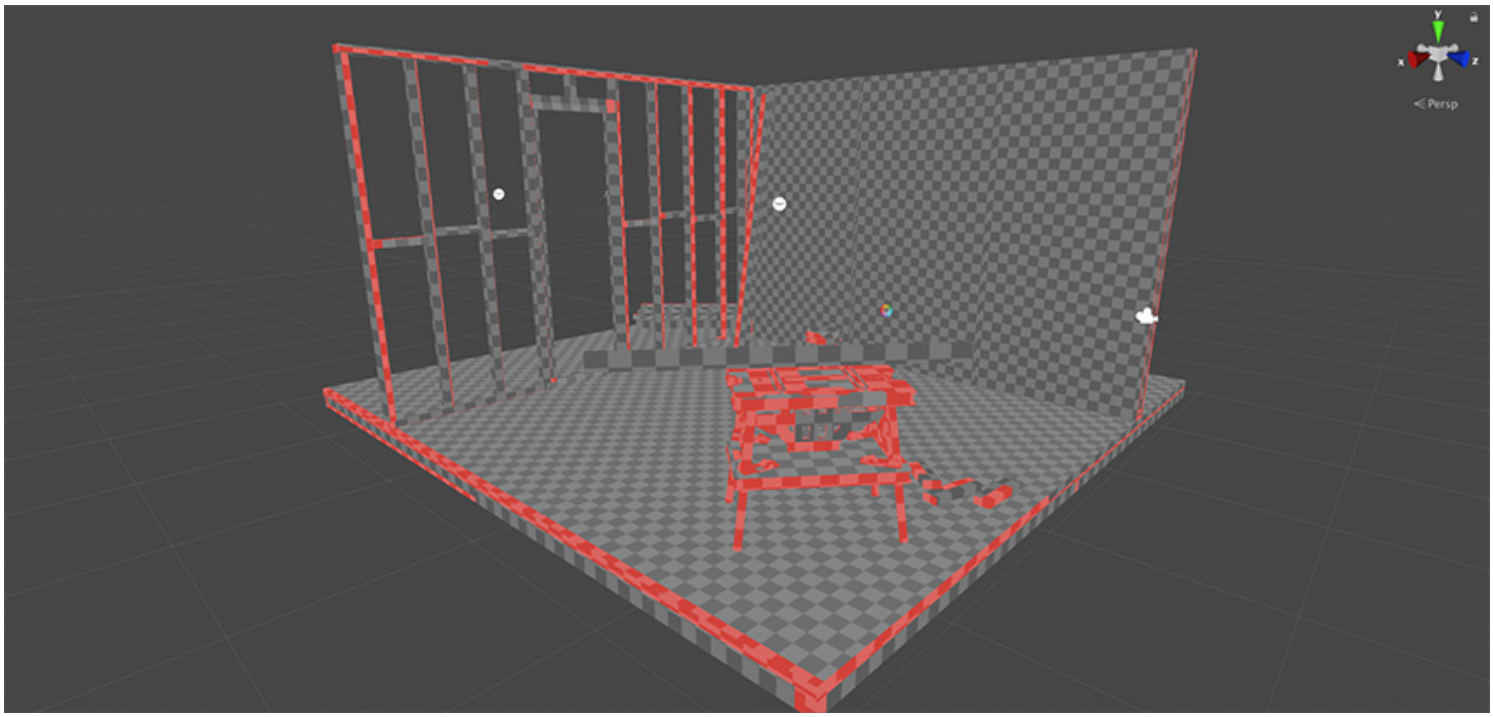
Texel Validity

This mode shows which texels are marked invalid because they mostly “see” backfaces. During lightmap baking, Unity emits rays from each texel. If a significant portion of a texel’s rays are hitting backface geometry, this texel is marked invalid. This is because the texel should not be able to see the backfaces in the first place. Unity handles this by replacing invalid texels with valid neighbors. You can adjust this behaviour using the **Backface Tolerance** parameter (**LightmapParameters** > **General GI**).



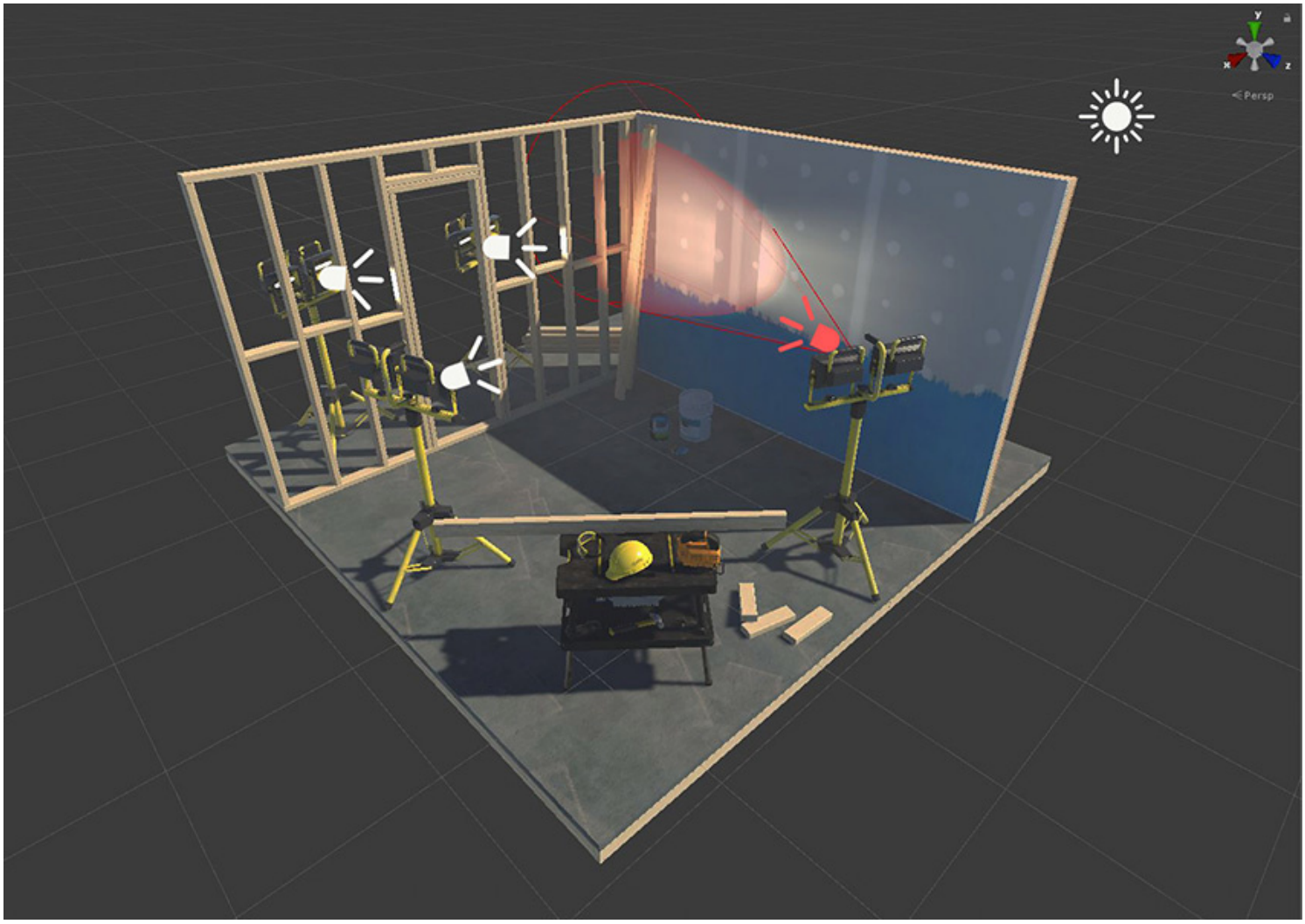
UV Overlap

If lightmap charts are too close together in UV space, the **pixel** values inside them might bleed into one another when the lightmap is sampled by the GPU. This can lead to unexpected artifacts. This mode allows you to identify texels that are too close to texels in other charts. This is useful when you want to troubleshoot your UV issues.



Light Overlap

This mode allows you to see if all static lights have been baked to the shadowmask. If an area of the level is lit by more than four static lights, the exceeding lights will fallback to fully baked and be displayed in red. Relevant for this calculation is not the actual lit surface, but the intersection of the light sources' volumes. So even though in the screenshot below it looks as if the colored spots on the mesh do not overlap, the cones of the four spotlights end up overlapping below the ground plane along with the directional light.



2018–03–28 Page amended with limited editorial review

Updated in 5.6

Updated in 2018.1

Lighting Data Asset

Leave feedback

Lightmap Snapshot was renamed to Lighting Data **Asset** in Unity 5.3.

You generate the Lighting Data Asset by pressing the Build button in the Lighting window. Your lighting will be loaded from that asset when you reload the **scene**.

The Lighting Data Asset contains the GI data and all the supporting files needed when creating the lighting for a scene. The asset references the renderers, the realtime **lightmaps**, the baked **lightmaps**, **light probes**, **reflection probes** and some additional data that describes how they fit together. This also includes all the **Enlighten** data needed to update the realtime global illumination in the Player. The asset is an Editor only construct so far, so you can't access it in the player. When you change the scene, for instance by breaking a **prefab** connection on a lightmap static object, the asset data will get out of date and has to be rebuilt.

Currently, this file is a bit bloated as it contains data for multiple platforms - we will fix this. Also we are considering adding some **compression** for this data.

The intermediate files that are generated during the lighting build process, but is not needed for generating a Player build is not part of the asset, they are stored in the GI Cache instead.

The build time for the Lighting Data Asset can vary. If your GI Cache is fully populated i.e. you have done a bake on the machine before (with the scene in its current state) it will be fast. If you are pulling the scene to a machine with a blank cache or the cache data needed has been removed due to the cache size limit, the cache will have to be populated with the intermediate files first which requires the precompute and bake processes to run. These steps can take some time.

Lightmap Directional Modes

Leave feedback

There are two **Directional Modes** available for light maps: **Directional** and **Non-Directional**. To set the **Directional Mode** for a light map, open the Lighting window (**Window** > **Lighting** > **Settings**), click **Scene**, navigate to the **Lightmapping Settings**, ensure the **Lightmapper** is set to **Enlighten**, and use the **Directional Mode** drop-down menu. Both are available as real-time and baked lightmaps.

Directional light maps store more information about the lighting environment than **Non-Directional** light maps. **Shaders** can use that extra data about incoming light to better calculate outgoing light, which is how materials appear on the screen. This happens at the cost of increased texture memory usage and shading time.

Non-directional: flat diffuse. This mode uses just a single **lightmap**, storing information about how much light does the surface emit, assuming it's purely diffuse. Objects lit this way will appear flat (normalmaps won't be used) and diffuse (even if the material is specular), but otherwise will be lit correctly. These barrels are using baked lightmaps. The only detail definition comes from a **reflection probe** and an occlusion map.

Directional: normal mapped diffuse. This mode adds a secondary lightmap, which stores the incoming dominant light direction and a factor proportional to how much light in the first lightmap is the result of light coming in along the dominant direction. The rest is then assumed to come uniformly from the entire hemisphere. That information allows the material to be normal mapped, but it will still appear purely diffuse.

Performance

Directional mode uses twice as much texture memory as **Non-directional** mode and has a slightly higher shading cost.

Non-directional: one texture, one texture sample, a few extra shader instructions.

Directional: two textures, two texture samples, a few more extra shader instructions.

Real-time lightmaps take advantage of the same approach, and are subject to the same shading quality/cost tradeoffs.

The BRDF that is actually used for indirect light (the indirect part of baked) is a slightly less expensive version. `UNITY_BRDF_PBS_LIGHTMAP_INDIRECT` is defined in *UnityPBSLighting.cginc*.

Specular lighting on light maps

To achieve specular light on lightmap static **assets**, use the Light Modes Shadowmask or Distance Shadowmask on **Baked lights** . This ensures the light is real-time and high quality. See documentation on Light Modes for more information.)

2017–06–08 Page published with limited editorial review

Direct Specular removed in 5.6

Light Modes added in 5.6

Lightmaps: Technical information

Leave feedback

Unity stores **lightmaps** with different compressions and encoding schemes, depending on the target platform and the compression setting in the Lighting Window.

Encoding schemes

Unity projects can use two techniques to encode baked light intensity ranges into low dynamic range textures when this is needed:

RGBM encoding. RGBM encoding stores a color in the **RGB** channels and a multiplier (**M**) in the alpha channel. The range of RGBM lightmaps goes from 0 to 34.49($5^{2.2}$) in linear space, and from 0 to 5 in gamma space.

Double Low Dynamic Range (dLDR) encoding. dLDR encoding is used on mobile platforms by simply mapping a range of [0, 2] to [0, 1]. Baked light intensities that are above a value of 2 will be clamped. The decoding value is computed by multiplying the value from the lightmap texture by 2 when gamma space is used, or 4.59482($2^{2.2}$) when linear space is used. Some platforms store lightmaps as dLDR because their hardware compression produces poor-looking artifacts when using RGBM.

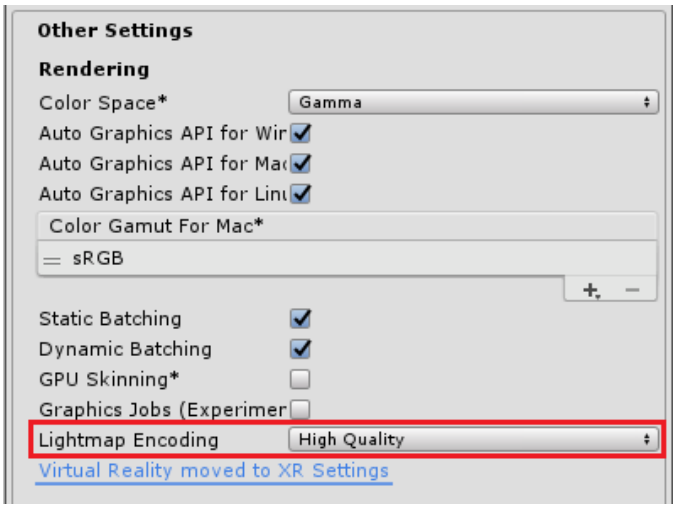
When Linear Color Space is used, the lightmap texture is marked as sRGB and the final value used by the **shaders** (after sampling and decoding) will be in Linear Color Space. When Gamma Color Space is used, the final value will be in Gamma Color Space.

Note: When encoding is used, the values stored in the lightmaps (GPU texture memory) are always in Gamma Color Space.

The **Decode Lightmap** shader function from the *UnityCG.cginc* shader include file handles the decoding of lightmap values after the value is read from the lightmap texture in a shader.

HDR lightmap support

HDR lightmaps can be used on PC, Mac & Linux Standalone, **Xbox One**, and PlayStation 4. The **Player Settings inspector** has a **Lightmap Encoding** option for these platforms, which controls the encoding/compression of the lightmaps.



Choosing **High Quality** will enable **HDR** lightmap support, whereas **Normal Quality** will switch to using **RGBM** encoding.

When lightmap **Compression** is enabled in the Lighting Window, the lightmaps will be compressed using the **BC6H** compression format.

Advantages of High Quality (BC6H) lightmaps

HDR lightmaps don't use any encoding scheme to encode lightmap values, so the supported range is only limited by the 16-bit floating point **texture format** that goes from 0 to 65504.

BC6H format quality is superior to DXT5 + RGBM format encoding, and it doesn't produce any of the color banding artifacts that RGBM encoding has.

Shaders that need to sample HDR lightmaps are a few ALU instructions shorter because there is no need to decode the sampled values.

BC6H format has the same GPU memory requirements as DXT5.

Here is the list of encoding schemes and their **texture compression** formats per target platform:

Target platform	Encoding	Compression - size (bits per pixel)
Standalone(PC, Mac, Linux)	RGBM / HDR	DXT5 / BC6H - 8 bpp
Xbox One	RGBM / HDR	DXT5 / BC6H - 8 bpp
PlayStation4	RGBM / HDR	DXT5 / BC6H - 8 bpp
WebGL 1.0 / 2.0	RGBM	DXT5 - 8 bpp
iOS	dLDR	**PVRTC** RGB - 4 bpp
tvOS	dLDR	**ASTC** 4x4 block RGB - 8 bpp
Android*	dLDR	ETC1 RGB - 4 bpp
Samsung TV	dLDR	ETC1 RGB - 4 bpp
Nintendo 3DS	dLDR	ETC1 RGB - 4 bpp
Nintendo 3DS	dLDR	ETC1 RGB - 4 bpp

*If the target is **Android** then you can override the default texture compression format from the **Build Settings** to one of the following formats: DXT1, PVRTC, **ATC**, ETC2, ASTC. The default format is **ETC**.

Precomputed real-time Global Illumination (GI)

The inputs to the GI system have a different range and encoding to the output. Surface albedo is 8-bit unsigned integer RGB in gamma space and emission is 16-bit floating point RGB in linear space. For advice on providing custom inputs using a meta pass, see documentation on Meta pass.

The irradiance output texture is stored using the RGB9E5 shared exponent floating point format if the graphics hardware supports it, or RGBM with a range of 5 as fallback. The range of RGB9E5 lightmaps is [0, 65408]. For details on the RGB9E5 format, see Khronos.org: EXT_texture_shared_exponent.

See also:

Texture Importer Override
Texture Types
Global Illumination
Texture Types
Global Illumination

2017–09–18 Page amended with no editorial review

Baked lightmaps added in Unity 2017.2

HDR lightmap support added in Unity 2017.3

Material properties and the GI system

Leave feedback

The way an object looks is defined by its **Shader**.

Legacy and current Shader mappings

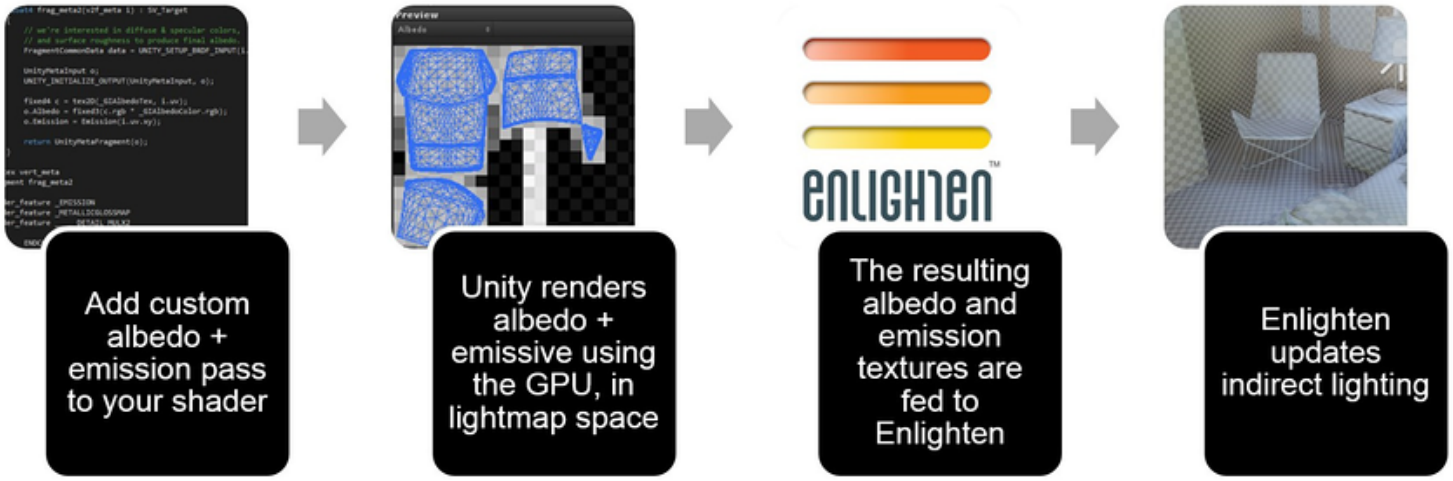
Shader mappings in Unity versions 3 and 4 work in a different way to Shader mappings in Unity 5 onwards. The legacy Shader mappings are still supported in Unity 5 onwards. See Legacy material mappings, below.

Unity versions 3.x and 4.x used a simple mapping from material properties to **lightmapper** material properties. It worked for common cases but was based on naming conventions, tags and strings. You couldn't do any custom surface properties as it was effectively hard coded to behave in a certain way. Unity version 5.0 onwards has flexible Shader mappings.

Meta pass (Unity 5.0 onwards)

Albedo and emissive are rendered using a special Meta Shader pass. **Lightmap** static **GameObjects** are rendered in **lightmap** space using the GPU. This means how the GameObject looks on screen and how it looks to the lightmapper are separate, so you can customize the Shaders.

The Meta pass decouples the albedo and emissive, which is used to compute Global Illumination (GI) during regular Shader passes. This allows you to control GI without affecting the Shader used for real-time **rendering**. The **standard Shader** contains a Meta pass by default. Global Illumination is managed in Unity by a piece of middleware called **Enlighten**.



Meta pass flow

The Meta pass is how the Unity Editor handles albedo for metallic surfaces internally. Enlighten handles diffuse transport and uses surface albedo on each bounce. Metallic surfaces with black (or almost black) albedo do not bounce any light. The Shader pass that renders albedo biases it towards a brighter color with the hue of the metal. Dielectric materials (wood, plastic, plastic, stone, concrete, leather, skin) have white specular reflectance. Metals have spectral specular reflectance.

Note: Using the Meta pass is not as fast as DynamicGI.SetEmissive, but it is more flexible because you are not limited to a single color.

Legacy material mappings

The built-in legacy Shaders in Unity version 5.0 and newer contain a Meta pass already. If you are upgrading projects from Unity versions before 5.0, you should add a Meta pass. See Example Shader with a Meta pass, below, to learn how.

Custom RGB transparency

You can add custom color-based RGB transparency by adding a texture property called `_TransparencyLM` to a Shader. In this case, the standard behavior is dropped and only the values of this texture are used to evaluate the transmission through the material. This is useful when you want to create color-based transparency that is independent of the material color or albedo texture.

To create custom transmission behavior, add the following line to a Shader and assign a Texture:

```
_TransparencyLM ("Transmissive Texture", 2D) = "white" {}
```

Note: Unity detects certain legacy Shaders by the Shader's properties and path/name keywords, such as `Transparent`, `Tree`, `Leaf`, `Leaves`.

Example Shader with a Meta pass

The Shader below allows for specifying a custom albedo color and Texture just for the GI system.

```
Shader "Custom/metaPassShader"{

    Properties {
        _Color ("Color", Color)=(1,1,1,1)
        _MainTex ("Albedo (RGB)",2D)="white"{}
        _Glossiness ("Smoothness", Range(0,1))=0.5
        _Metallic ("Metallic", Range(0,1))=0.0

        _GIAlbedoColor ("Color Albedo (GI)", Color)=(1,1,1,1)
        _GIAlbedoTex ("Albedo (GI)",2D)="white"{}
    }

    __SubShader__ {
    // ------------------------------------------------------------------
    // Extracts information for lightmapping, GI (emission, albedo, ...)
    // This pass is not used during regular rendering.
        Pass
        {
            Name "META"
            Tags {"LightMode"="Meta"}
            Cull Off
            CGPROGRAM

            #include"UnityStandardMeta.cginc"

            sampler2D _GIAlbedoTex;
            fixed4 _GIAlbedoColor;
            float4 frag_meta2 (v2f_meta i): SV_Target
            {
                // We're interested in diffuse & specular colors
                // and surface roughness to produce final albedo.

                FragmentCommonData data = UNITY_SETUP_BRDF_INPUT (i.uv);
                UnityMetaInput o;
                UNITY_INITIALIZE_OUTPUT(UnityMetaInput, o);
                fixed4 c = tex2D (_GIAlbedoTex, i.uv);
```

```
                o.Albedo = fixed3(c.rgb * _GIAlbedoColor.rgb);
                o.Emission = Emission(i.uv.xy);
                return UnityMetaFragment(o);
            }

            #pragma vertex vert_meta
            #pragma fragment frag_meta2
            #pragma shader_feature _EMISSION
            #pragma shader_feature _METALLICGLOSSMAP
            #pragma shader_feature ___ _DETAIL_MULX2
            ENDCG
        }

        Tags {"RenderType"="Opaque"}
        __LOD__ 200

        CGPROGRAM
        // Physically-based Standard lighting model, and enable shadows on all light types
        #pragma surface surf Standard fullforwardshadows nometa
        // Use Shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        void surf (Input IN,inout SurfaceOutputStandard o){
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex)* _Color;
            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }

    FallBack "Diffuse"
}
```

2017–10–31 Page amended with limited editorial review

Global Illumination UVs

Leave feedback

There are two sets of GI lightmaps: **Baked** and **Realtime**. How you define which set to use depends on whether you're working with environment lighting or specific lights:

Global illumination (environment lighting) can be set to **Realtime** or **Baked**. Go to **Window** > **Rendering** > **Lighting Settings** and choose an option from the **Ambient GI** drop-down menu.

Lights can be set to **Realtime**, **Baked** or **Mixed**. Go to the Inspector window and choose an option from the **Baking** drop-down menu.

Materials have emission controls that can be set to **Realtime** or **Baked**. See documentation on Standard Shader Material parameter emissions to learn more.

Lightmap Properties	
Baked	Baked lightmaps are mainly useful for lights which do not change at all during run time (for example, a lit streetlamp), and are therefore stored as a static rendering in the lightmap. Features include direct lighting, indirect lighting, and **ambient occlusion**.
Realtime	Real-time lightmaps are mainly useful for lights that are animated during runtime (for example, a flickering street lamp), and therefore need to be rendered in real time. Features include indirect lighting only, and typically low resolution. Direct light is not in the lightmap, but is rendered in real time.
Mixed	When you set a light to Mixed mode, it contributes to the baked lightmaps, and also gives direct real-time lighting to non-static objects.

You can use either one or both of these lightmap sets to light your **Scenes**. Your choice determines which lightmaps the light contributions and resulting GI is added to.

Visualising your UVs

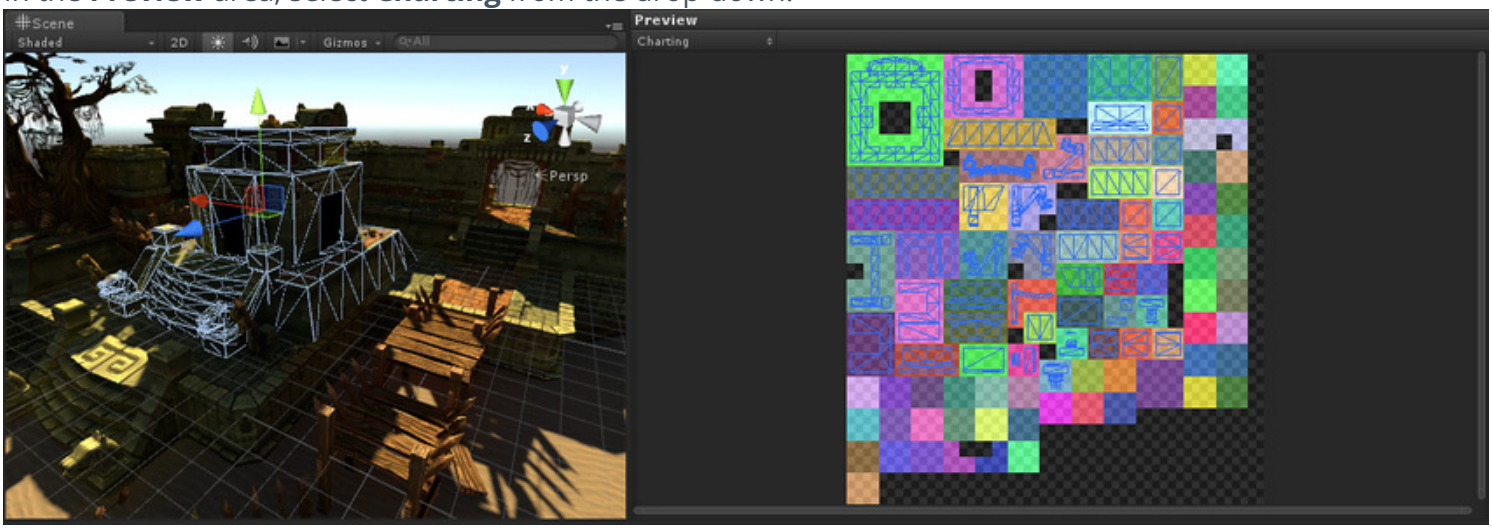
It is important to be able to view the UVs that are being used, and Unity has a visualization tool to help you with this. First, open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**) and tick the **Auto** checkbox at the bottom. This ensures that your bake and precompute are up-to-date, and outputs the data that is needed to view the UVs. Wait for the process to finish (this can take some time for large or complex Scenes).

Visualising real-time UVs

To see the precomputed real-time GI UVs:

Select a **GameObject** with a **Mesh** Renderer in your Scene
Open the Lighting window and select the **Object** tab

In the **Preview** area, select **Charting** from the drop-down.



This displays the UV layout for the real-time lightmap of the selected instance of this Mesh.

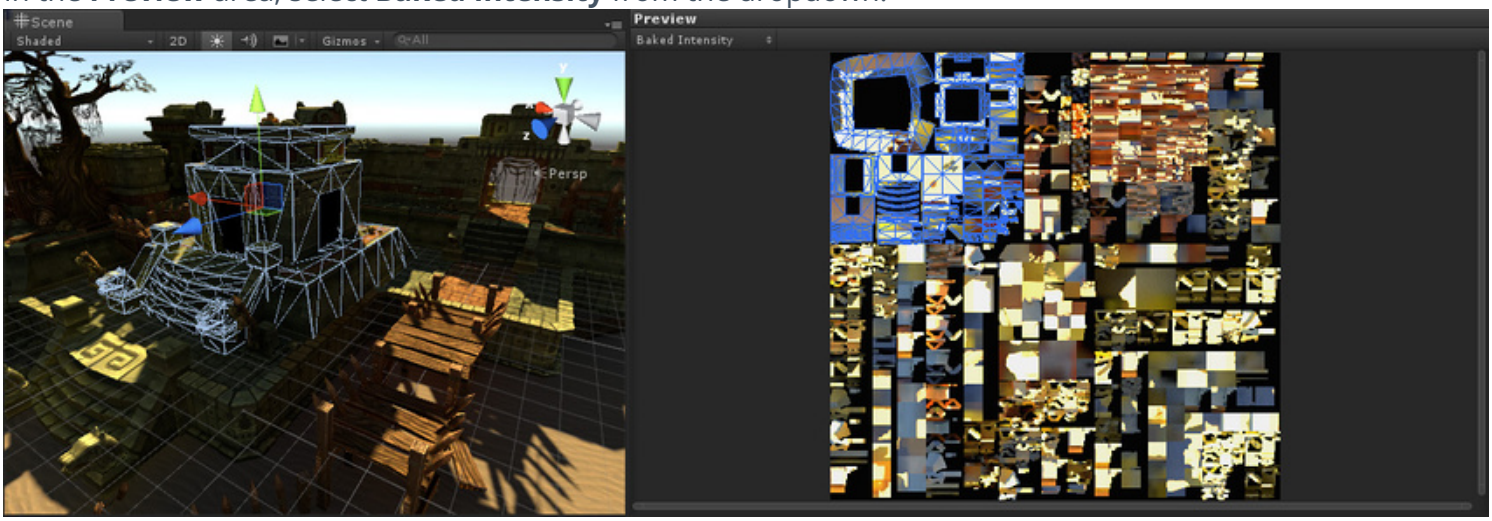
The charts are indicated by the different colored areas in the Preview (show in the image above on the right-hand side).
The UVs of the selected instance are laid over the charts, as a wireframe representation of the GameObject's Mesh.
Dark gray texels show unused areas of the lightmap.
Multiple instances can be packed into a real-time lightmap, so some of the charts you see might actually belong to other GameObjects.

NOTE: There is no direct correspondence in the grouping of instances between real-time and baked lightmaps. Two instances in the same real-time lightmap may also be in two different baked lightmaps, and vice versa.

Visualising baked UVs

To see the baked UVs:

Select an instance.
Open the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**) and select the **Object** tab.
In the **Preview** area, select **Baked Intensity** from the dropdown.



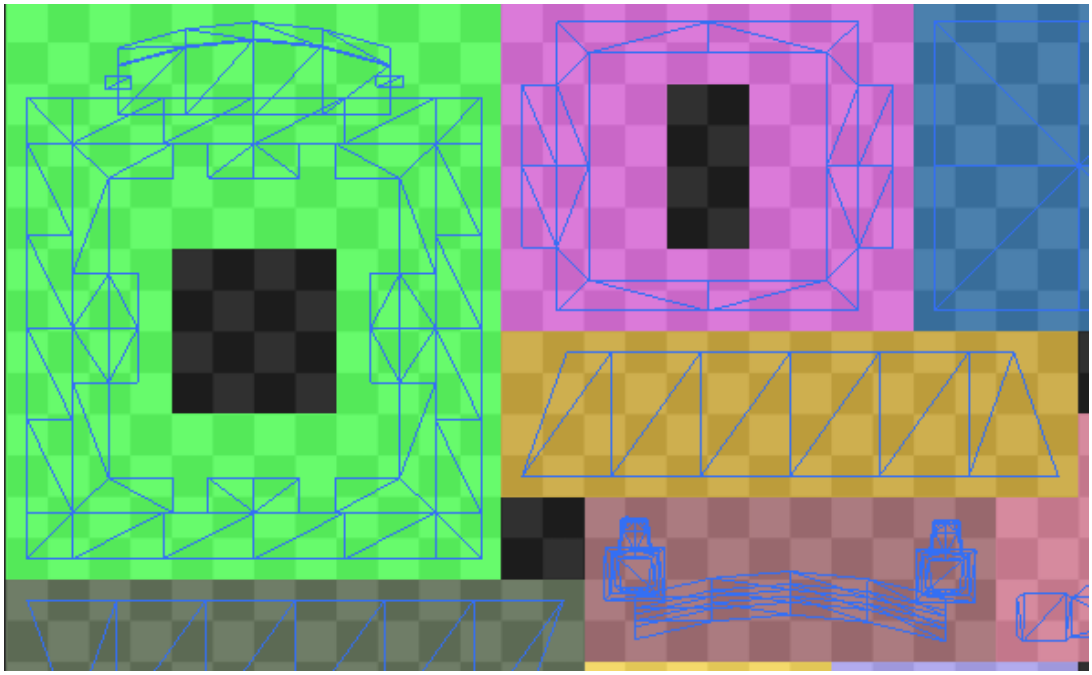
As you can see, the baked UVs are very different to the precomputed real-time UVs. This is because the requirements for baked and precomputed real-time UVs are different.

Real-time UVs

It is important to note that you can never get the same UVs for precomputed real-time GI as for baked GI, even if you tick **Preserve UVs**.

If you could, you would see heavy aliasing (such as light or dark edges) in unexpected places. This is because the resolution of real-time lightmaps is intentionally low, so that it is feasible to update them in real time. This doesn't affect the graphical quality, because it only stores indirect lighting, which is generally low frequency (meaning it does not usually have sudden changes in intensity or detailed patterns). The direct light and shadows are rendered separately using the standard real-time lighting and shadowmaps. Direct light is generally higher frequency (meaning it is more likely to have sudden changes in intensity or detailed patterns, such as sharp edges to shadows) and therefore requires higher resolution lightmaps to capture this information.

Low resolution lightmaps can create bleeding issues, caused when charts share texels. This has a detrimental effect on the quality of the lighting, but is solved by repacking the UV charts to ensure a half-pixel boundary around them. This way you are never sampling *across* charts (at the most detailed mip), even with bilinear **interpolation**. The other benefit of charts with a guaranteed half-pixel boundary is that you can place charts right next to each other, saving lightmap space.



In summary, UVs used for precomputed realtime GI lightmaps are always repacked.

Because repacking guarantees a half-pixel boundary around the charts, the UVs are dependent on the scale and lightmap resolution of the instance. If you scale up the UVs to get a higher resolution lightmap, you are no longer guaranteed this half-pixel boundary. The UVs are packed individually, with the scale and resolution of the instance taken into account. The real-time UVs are therefore *per instance*. Note that if you have 1000 objects with the same scale and resolution, they share the UVs.

2017–07–04 Page published with limited editorial review

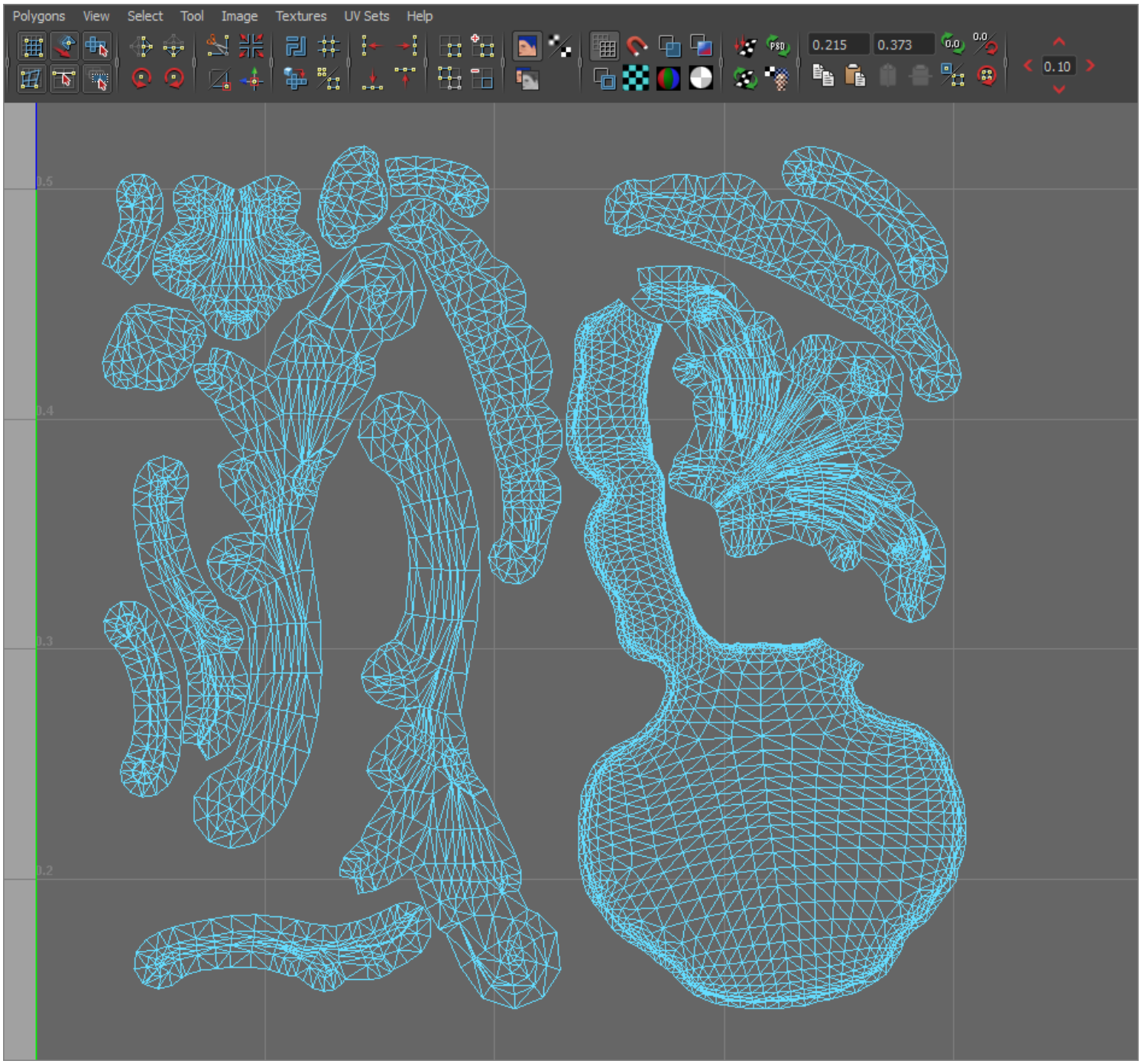
2017–07–04 Documentation update only, no change to Unity functionality

Importing UVs from Autodesk® Maya® to Unity

Leave feedback

Autodesk® Maya® is a 3D computer animation software by Autodesk, with powerful modeling, **rendering**, simulation, texturing and animation tools for visual effects artists, modellers and animators (see www.autodesk.co.uk). It is often used by Unity developers for advanced graphics work, which is then imported into Unity. It's important to note that when importing from Autodesk® Maya®, your UVs may not look exactly the same, even if you untick the **Optimize Realtime UVs** checkbox. This section explains why.

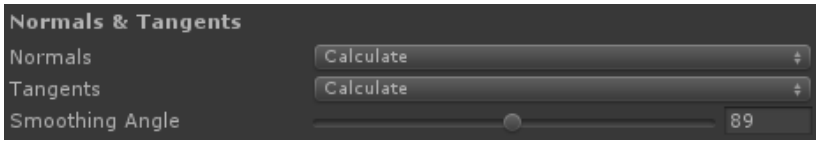
Because real-time UVs are repacked by **Enlighten**, it is very important to understand how UV charts are detected. By default, a chart is defined by a set of connected vertices. However, the DCC or Unity **Mesh** importer might introduce extra vertices in places where the **Mesh** has hard edges. These duplicated vertices create extra islands (unconnected groups) in your UVs. However, these cuts normally go unnoticed when you bake the **lightmap**, because the UVs are used directly and not repacked. The image below shows an example of this.



UVs in Autodesk® Maya®

A high smoothing angle does not preserve hard cuts in the model, and both the shading and GI look different as a result.

The Mesh Importer settings that relate to this are **Normals**, **Tangents**, and **Smoothing Angles**:

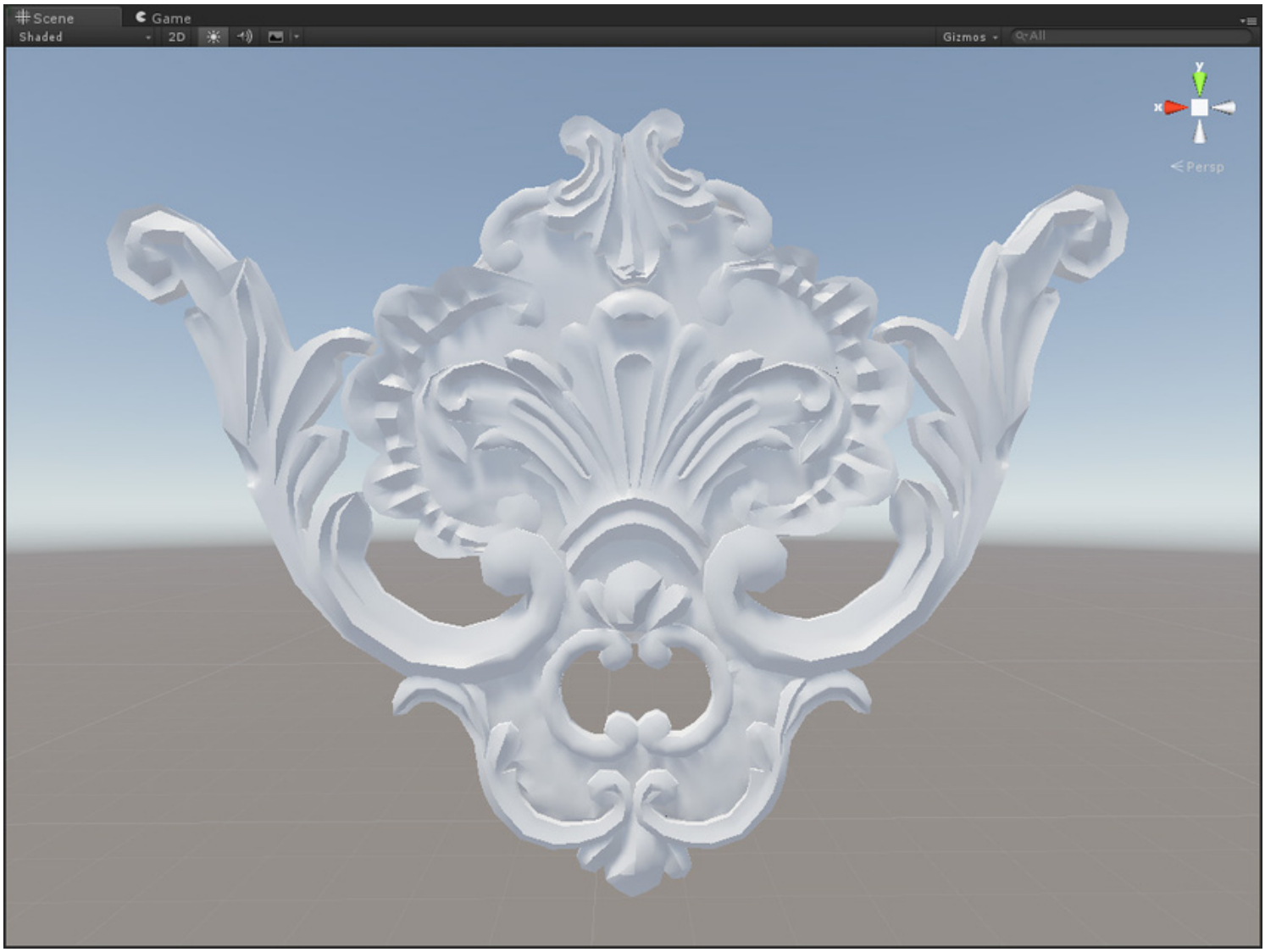


If you set **Normals** to **Calculate**, breaks are made wherever the angle between adjacent triangles exceeds the **Smoothing Angle** value.

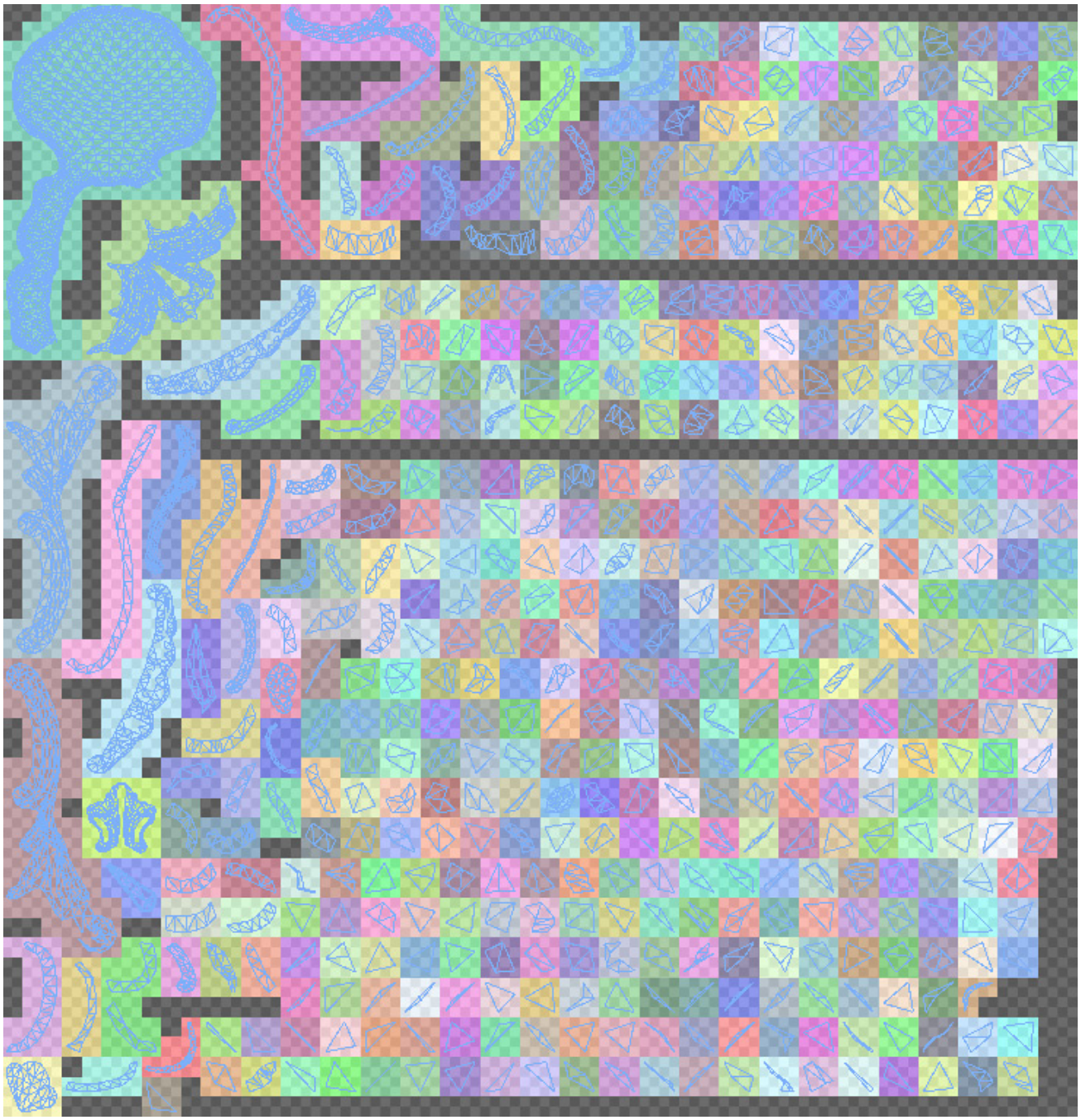
To avoid this, you can choose to author and import normals (see documentation on Normal maps to learn more about surface normals). In order to get good results with imported normals, you need to manually make the cuts along hard edges, and pay attention to how the DCC is inserting duplicate vertices. Otherwise, both GI and regular shading may have undesired lighting effects.

Example

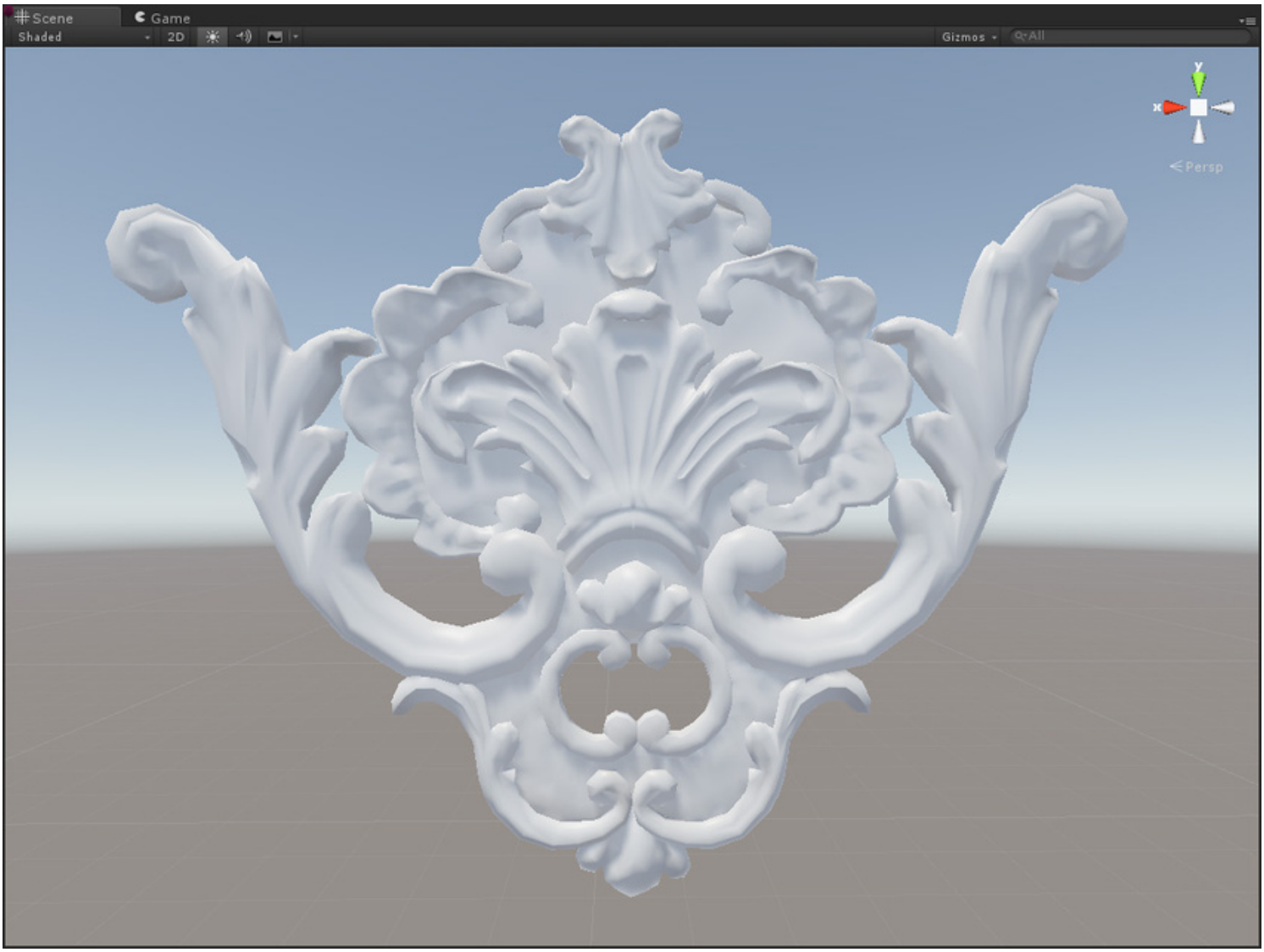
When packing with a 40-degree **Smoothing Angle**, the hard angles in the model are preserved, and extra charts are created:



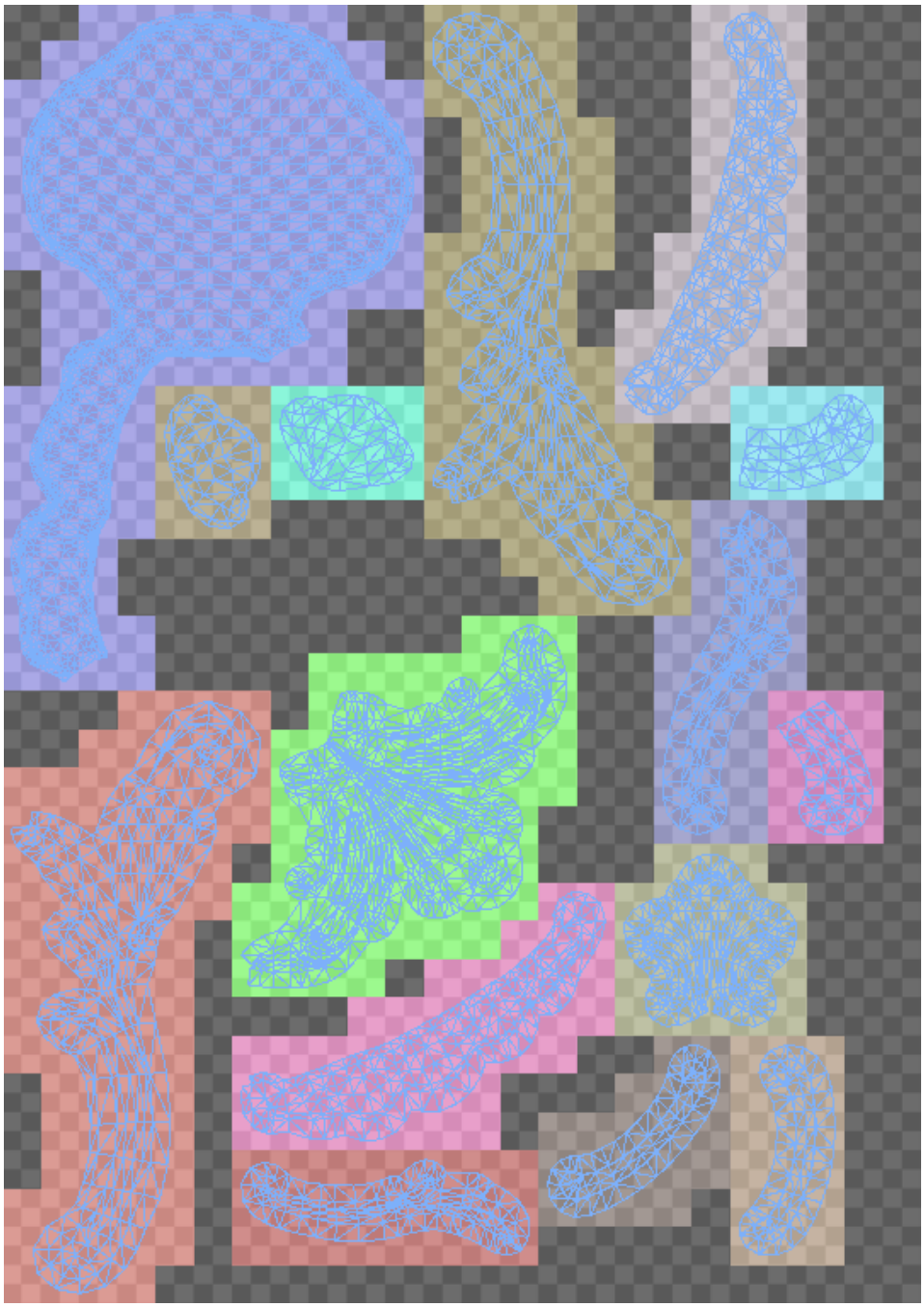
Asset source: Lee Perry Smith, VizArtOnline



If the **Smoothing Angle** is set to 180 degrees, no cuts are made, and the UVs are the same as they were in Autodesk® Maya®. The only difference is the chart packing:



Asset source: Lee Perry Smith, VizArtOnline



Optimizing Realtime UVs

The Mesh Renderer contains an option called **Optimize Realtime UVs**.

Optimize Realtime UVs enables Enlighten's UV optimization feature. Note that disabling this option does not allow the authored UVs to flow straight to Enlighten; repacking is still applied.

The feature is intended to optimize charting for real-time GI only. It does not affect baked UVs. Its purpose is to simplify the UV unwrap, which reduces the chart count (and thus texel count). This makes lighting more consistent across the model, makes the texel distribution more even, and avoids wasting texels on small details. The time taken to do the precompute phase is proportional to the number of texels you feed in. For example, a detailed tiled floor with separate charts for each tile takes up an

unnecessarily high number of texels, but joining them into a single chart results in far fewer texels. This works because the real-time lightmaps only store indirect lighting (meaning there are no sharp direct shadows).

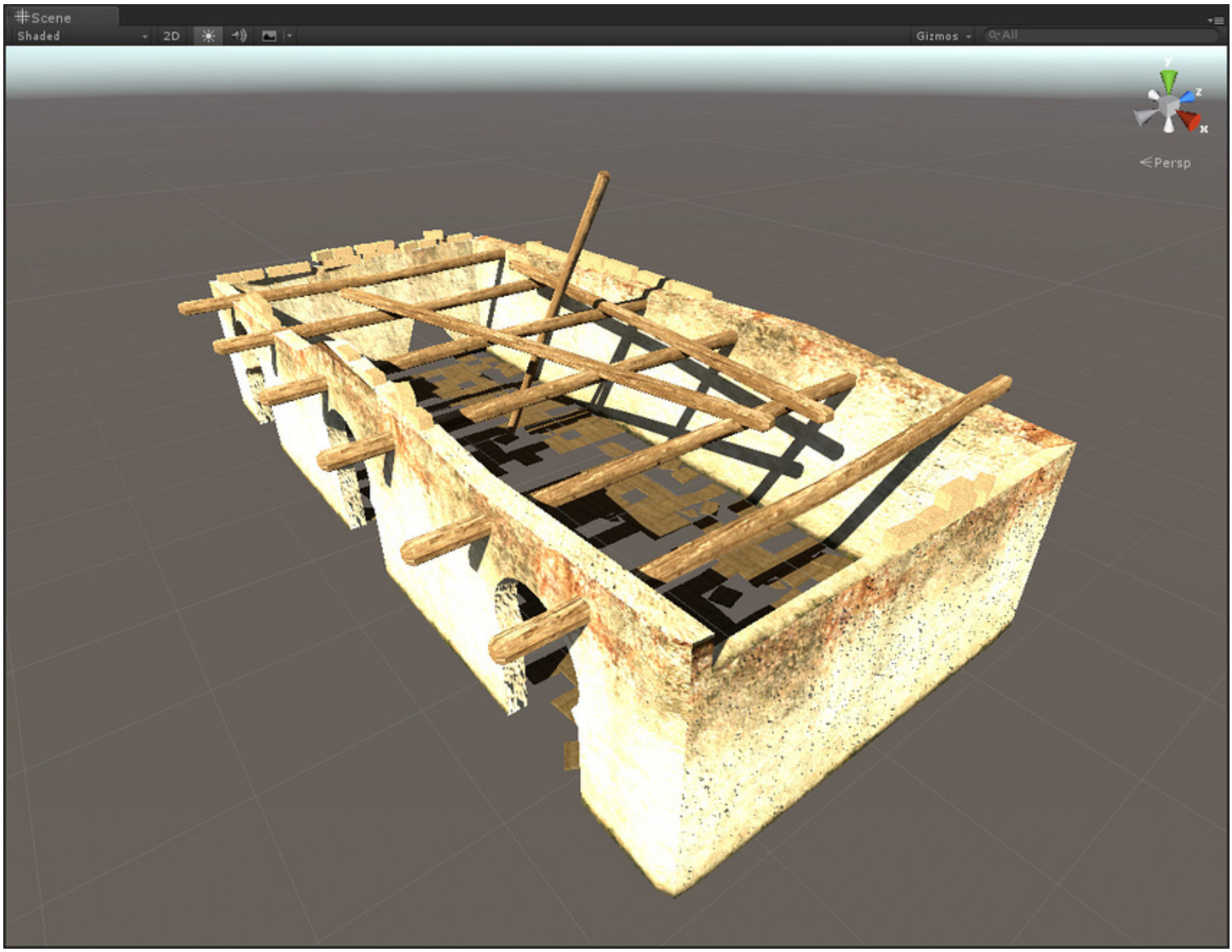
This process cannot alter the number of vertices in the model, so it cannot introduce breaks in the UVs where there already one present. This means the resulting chart layout is the same, but some of the charts might overlap or be merged in areas where it is unlikely to have a negative effect on the indirect lighting.

Use the settings to define when the charts are merged:

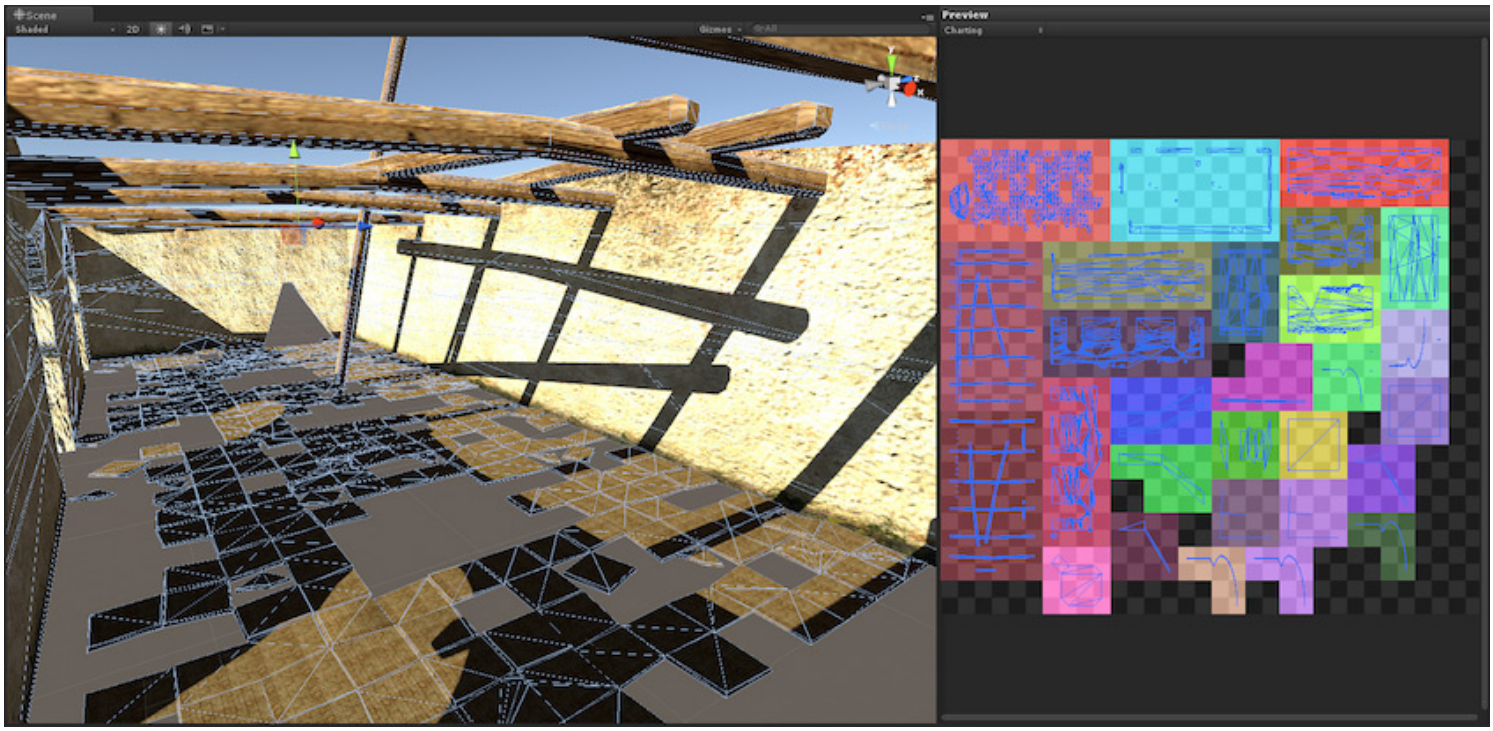
Max Distance: Charts are simplified if the worldspace distance between the charts is smaller than this value.
Max Angle: Charts are merged if the angle between the charts is smaller than this value.
These settings are intended to avoid merging charts when they are far apart or pointing in generally different directions.

Optimizing Realtime UVs: Example

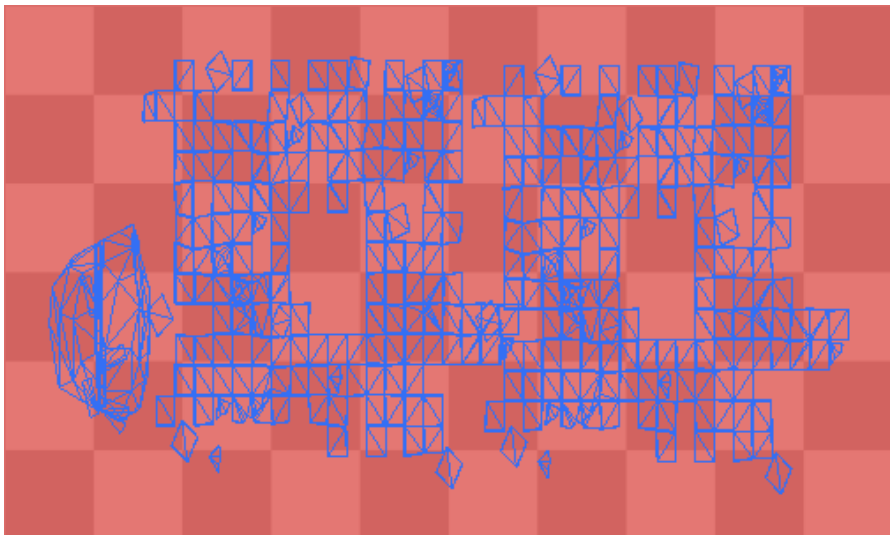
The following example uses the Desert Ruins **Asset** from the **Asset Store**:



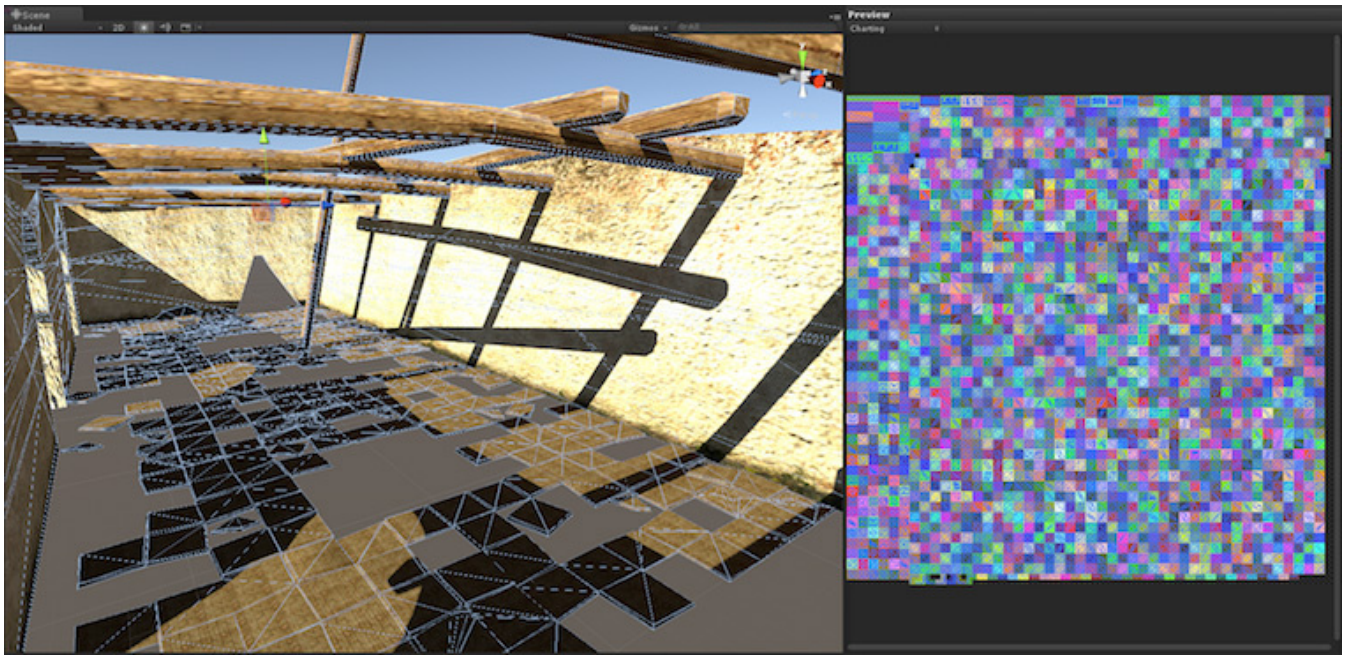
Asset source: DEXSOFT-Games
It uses default parameters, and the real-time lightmap resolution is 1 texel per unit. The model is approximately 9 units long. The image below shows the real-time UVs generated for this model using the Auto UV feature:



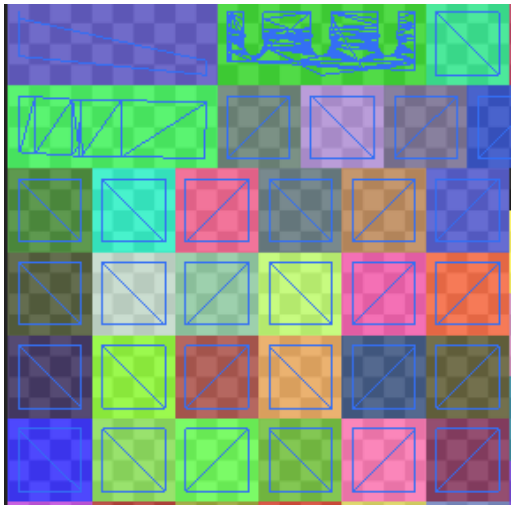
Note that the the tiles on the floor have been packed to a single chart, with an appropriate resolution for the chosen texel density and instance size:



When packed without the Auto UV feature, the generated UVs look like this:



This generates a large number of small charts, because the charts are split in the authored UVs supplied by the model. Because **Auto UVs** is not enabled, none of these charts can be merged, and each UV island is awarded a 4x4 **pixel** block of its own regardless of its size. The image below shows a subsection of the UVs:



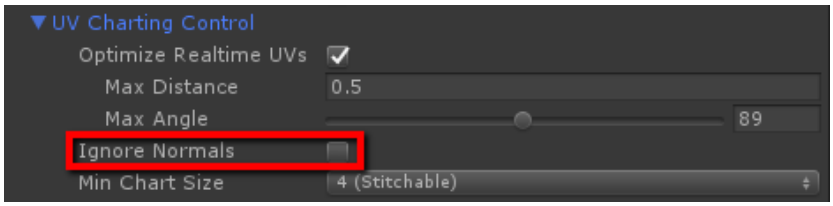
The wall sides still get a sensible resolution of 10x4 texels, but the small tiles gets a disproportionate 4x4 texels each. The reason that the minimum chart size is 4x4 is that we want to be able to stitch against the chart on all 4 sides and still get a lighting gradient across the chart.

Further Chart Optimization

There are two additional options for further optimizing the charting the UV layout:

Ignore Normals
Min Chart Size

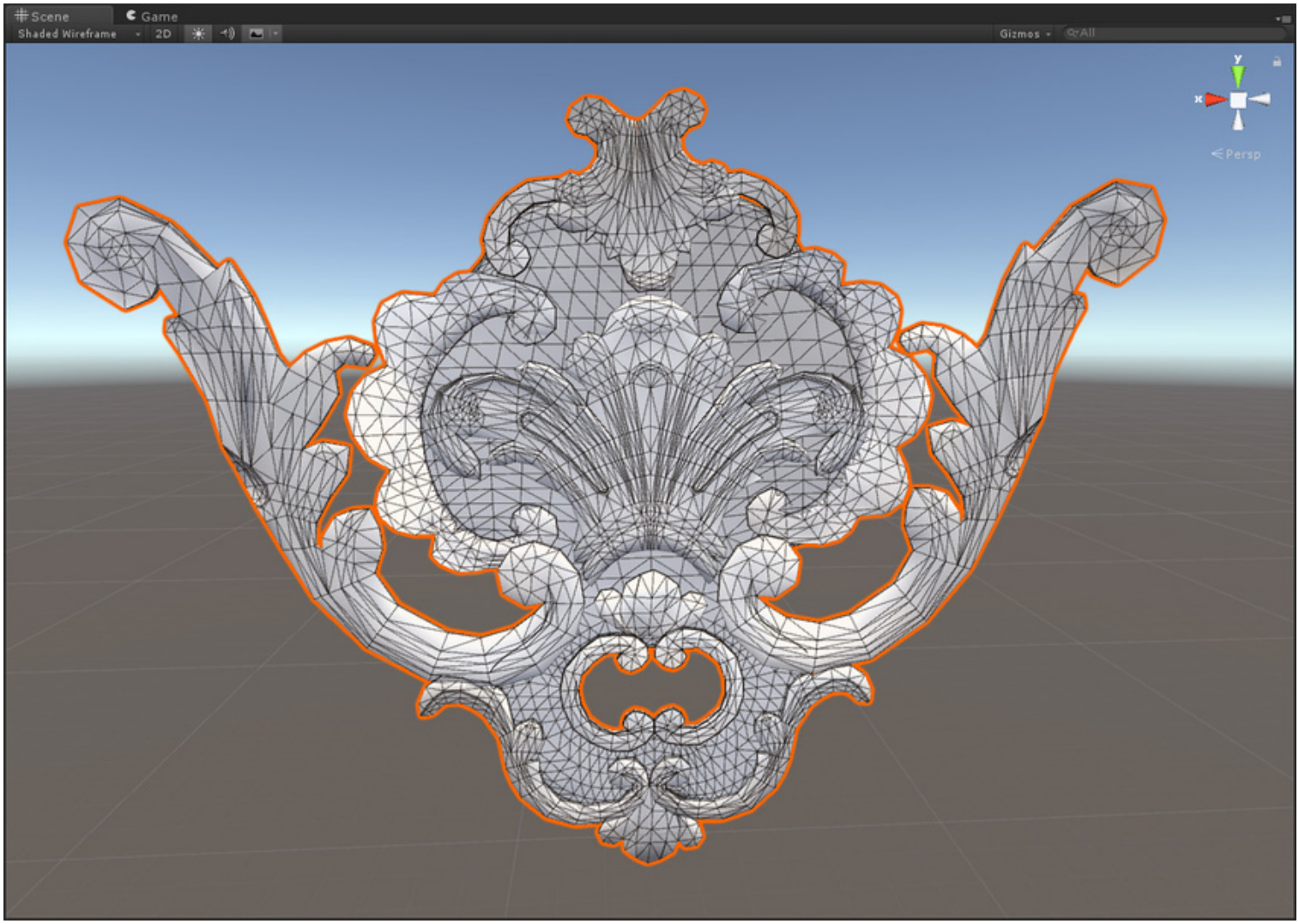
Ignore Normals



Tick the **Ignore Normals** checkbox to keep together any charts that have duplicated vertices due to hard normal breaks. A chart split might occur in Enlighten when the vertex position and the vertex lightmap UVs are the same, but the normals are different. For small details, multiple 4x4 texel charts to represent indirect lighting is too much, and affects precompute and baking performance. In these cases, enable **Ignore Normals**.

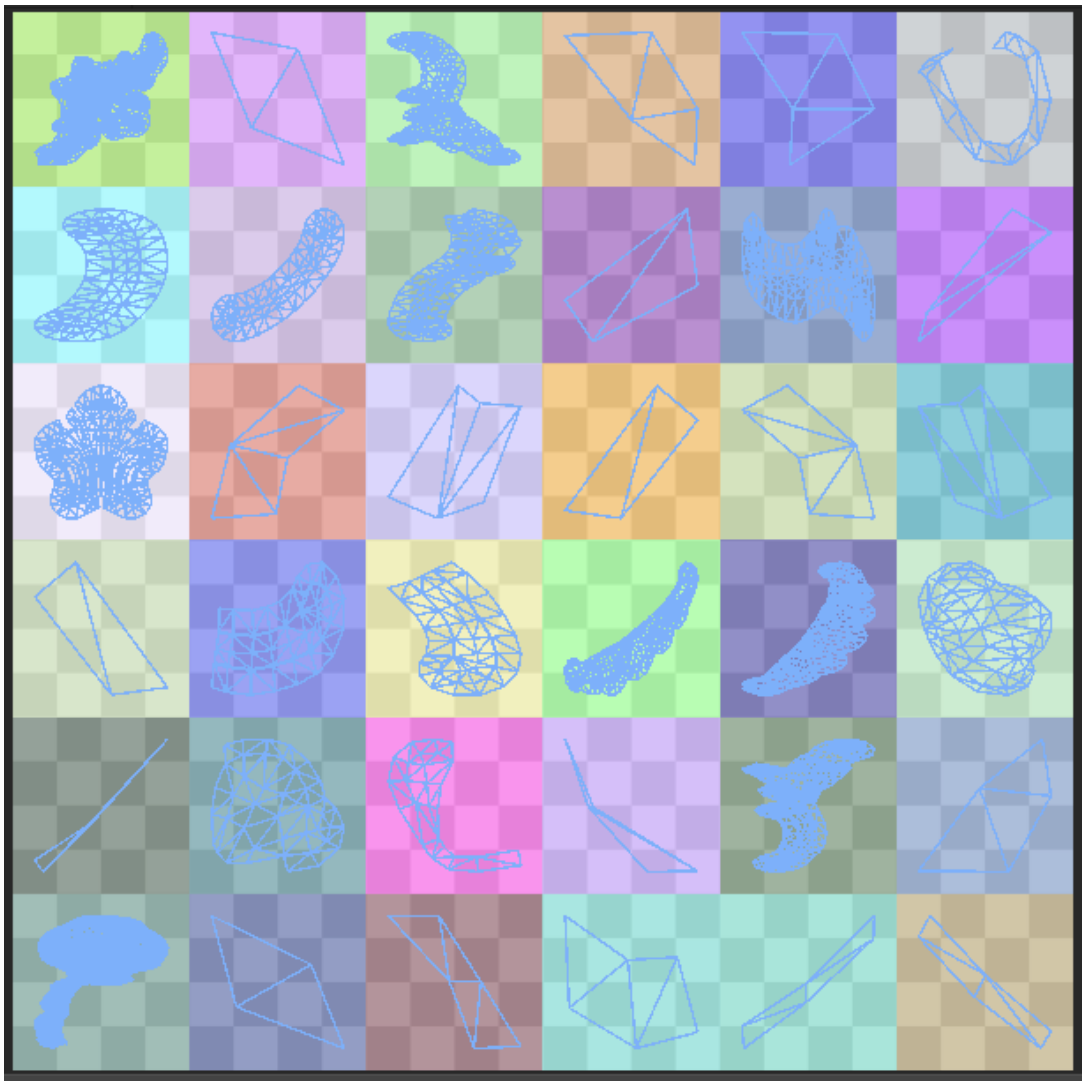
Example

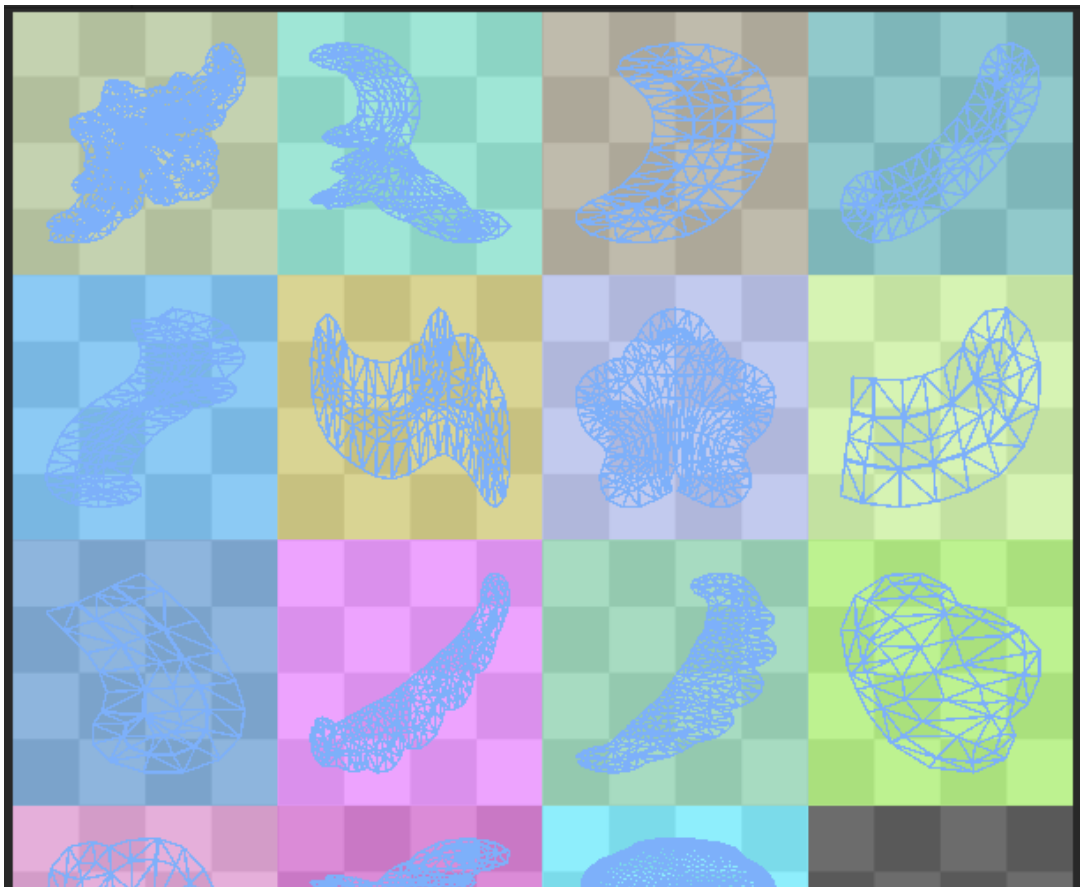
In the following example, **Optimize Realtime UVs** is disabled, to demonstrate the effect of **Ignore Normals** in isolation.



Asset source: Lee Perry Smith, VizArtOnline

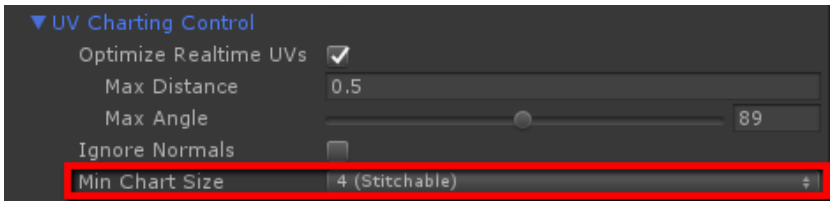
The image on the left shows the result without **Ignore Normals** enabled. The image on the right shows the result with **Ignore Normals** enabled.





When **Ignore Normals** is enabled, the 24x24 Enlighten unwrap is reduced to a 16x16 unwrap for this model.

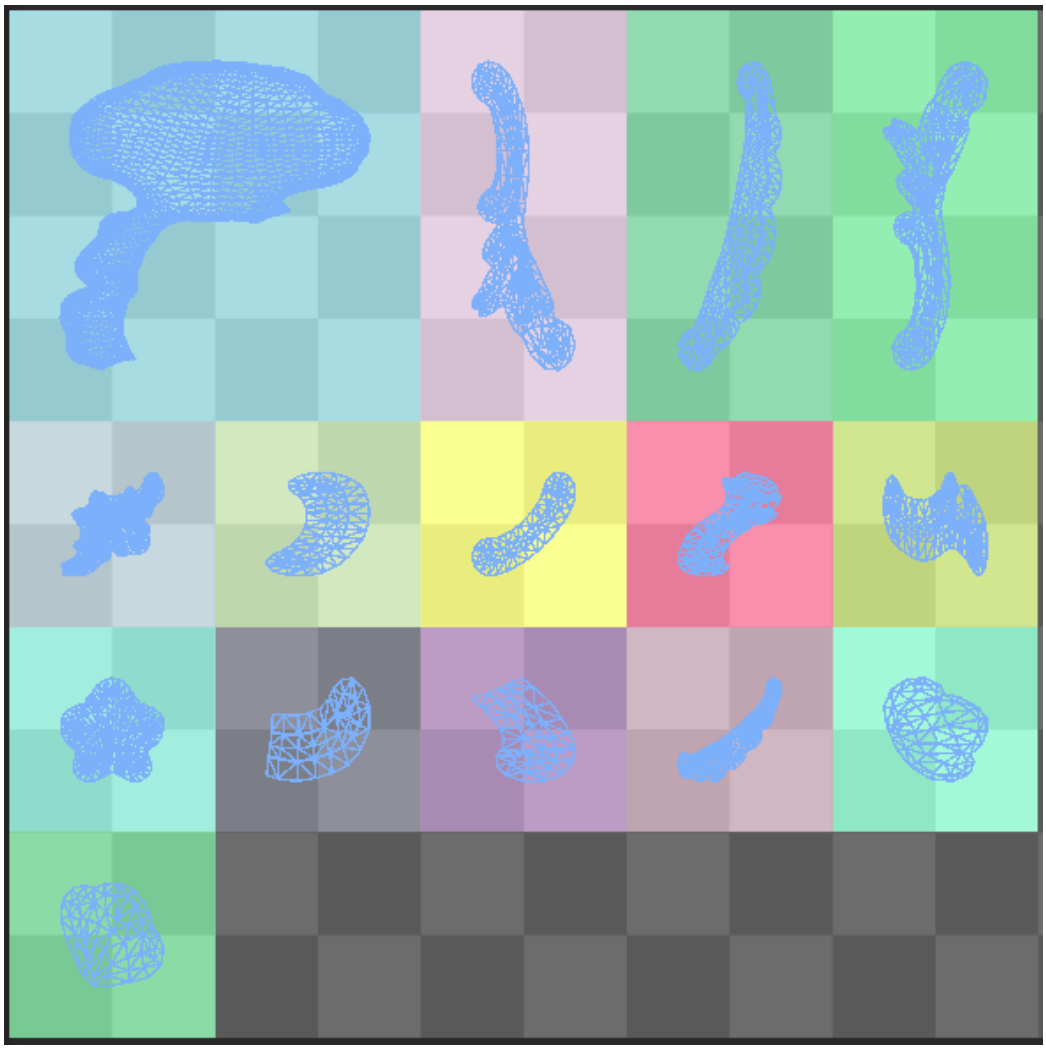
Min Chart Size



Min Chart Size removes the restriction of having a 4x4 minimum chart size. The stitching does not always work well, but for small details it is usually acceptable.

Example

In this example, **Min Chart Size** is set to **2 (Minimum)**.

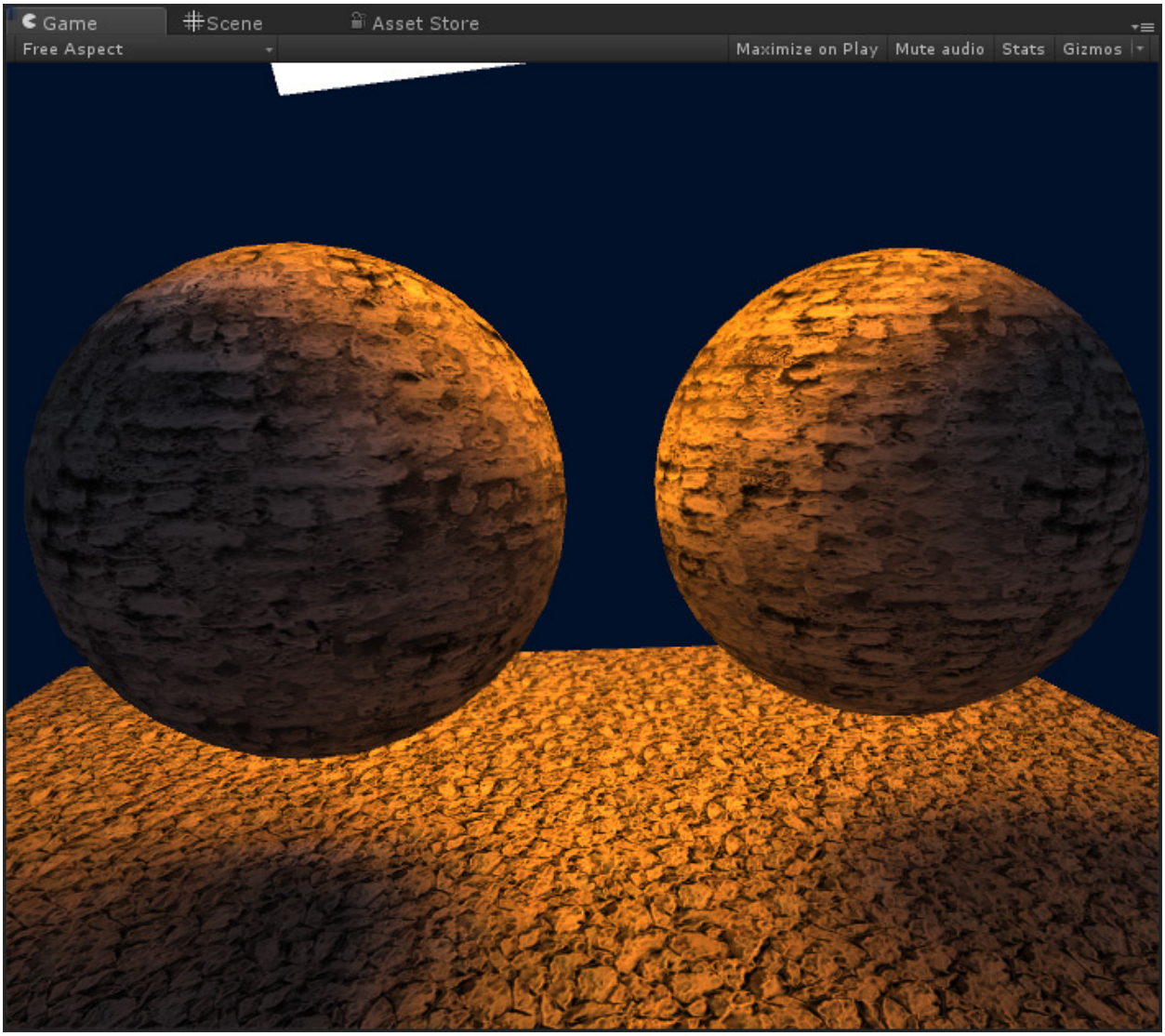


If you were to apply this **Min Chart Size** option and **Ignore Normals** to the model above, the unwrap reduces to 10x10.

Getting chart edges to stitch for realtime GI

The lightmaps that are set to **Realtime** support chart stitching. Chart stitching ensures that the lighting on adjacent texels in different charts is consistent. This is useful to avoid visible seams along the chart boundaries. In large texel sizes, the lighting on either side of a seam may be quite different. This difference is not automatically smoothed out by filtering, because the texels are not adjacent.

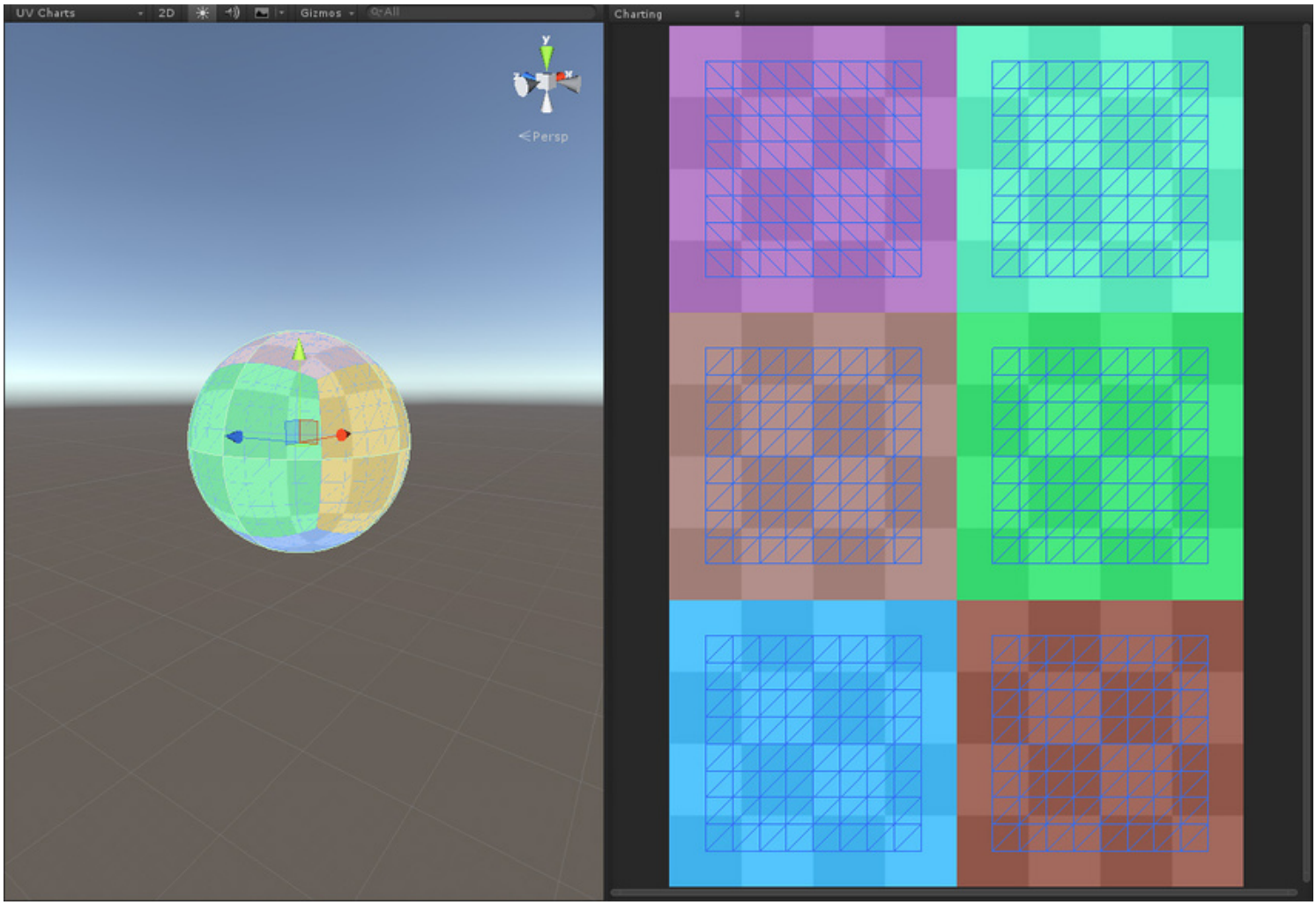
In this example, a seam is visible on the sphere on the right, even when textured, because it has not been stitched:

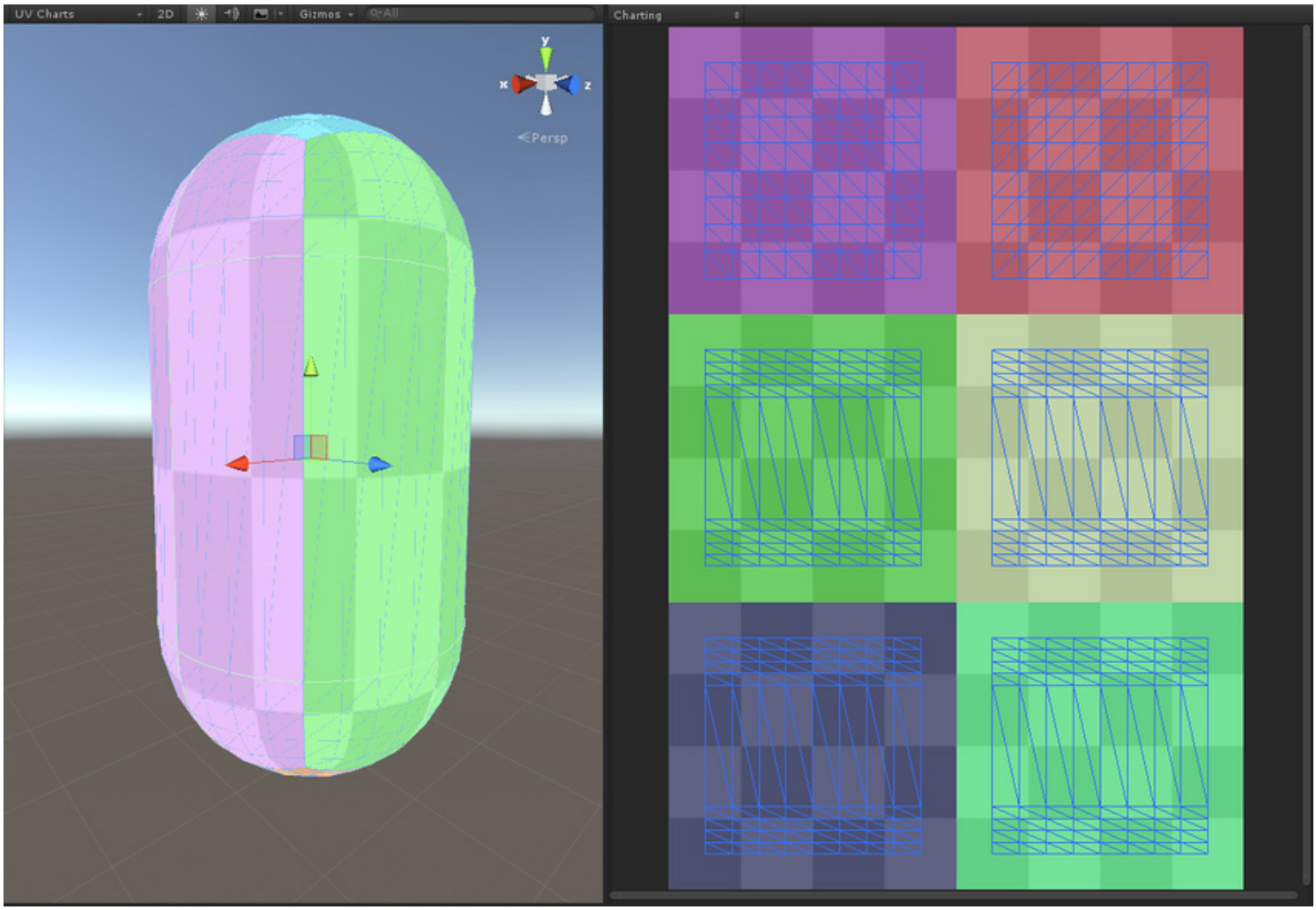


Stitching is on by default. If you think it is causing some unwanted issues, you can disable it; Apply Lightmap Parameters to the instance in question and untick the **Edge Stitching** checkbox.

For charts to stitch smoothly, edges must adhere to the following criteria:

Preserve UVs must be enabled, so that the charts are not simplified by the Auto UV feature.
The charts must be in the same Mesh.
The edges must share vertices.
The edges must be horizontal or vertical in UV space.
The edges must have the same number of texels (this usually follows from the two preceding criteria).
This is how Unity's built-in sphere, capsule and cylinder are authored. Notice how the charts line up:


UV Charts
2D
Gizmos
Charting
Persp



2017–07–04 Page published with limited editorial review

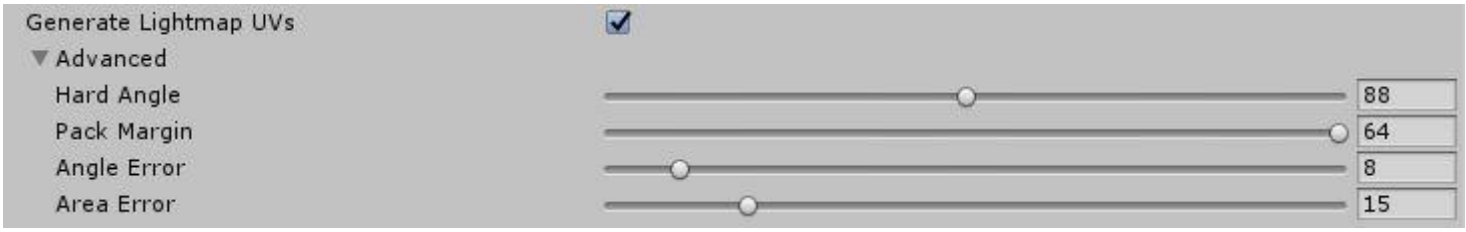
2017–07–04 Documentation update only, no change to Unity functionality

Generating Lightmap UVs

Leave feedback

Unity can unwrap your **Mesh** for you to generate **lightmap** UVs. To access the settings for generating **lightmap** UVs, open the Model's Import Settings, navigate to **Meshes**, and tick the **Generate Lightmap UVs** checkbox. This generates your lightmap UVs into UV2, if the channel is present. If the UV2 channel is not present, Unity uses primary UVs.

Click the **Advanced** foldout to open the settings.



Settings for Generate Lightmap UVs:

Property:	Function:
Hard Angle	The angle between neighboring triangles (in degrees) after which Unity considers it a hard edge and creates a seam. You can set this to a value between 0 and 180. This is set to 88 degrees by default. If you set this to 180 degrees, Unity considers all edges smooth, which is realistic for organic models. The default value (88 degrees) is realistic for mechanical models.
Pack Margin	The margin between neighboring charts (in pixels), assuming the Mesh takes up the entire 1024x1024 lightmap. You can set this to a value between 1 and 64. A larger value increases the margin, but also increases the amount of space the chart needs. This is set to 4 pixels by default. For more information, see Pack Margin, below.
Angle Error	The maximum possible deviation of UV angles from the angles in the source geometry (as a percentage from 0–100). This is set to 8% by default. This controls how different the triangles in UV space can be to the triangles in the original geometry. Usually this should be fairly low, to avoid artifacts when applying the lightmap.
Area Error	The maximum possible deviation of UV areas from the areas in the source geometry (as a percentage from 0–100). This is set to 15% by default. This controls how well Unity preserves the relative triangle areas. Increasing the value allows you to create fewer charts. However, increasing the value can change the resolution of the triangles, so make sure the resulting distortion does not deteriorate the lightmap quality.

You can also provide your own UVs for your lightmaps. A good UV set for lightmaps should adhere to the following rules:

It should be within the [0,1] x [0,1] UV space.

It should have a wide enough margin between individual charts. For more information, see documentation on UV overlap feedback.

It should not have any overlapping faces.

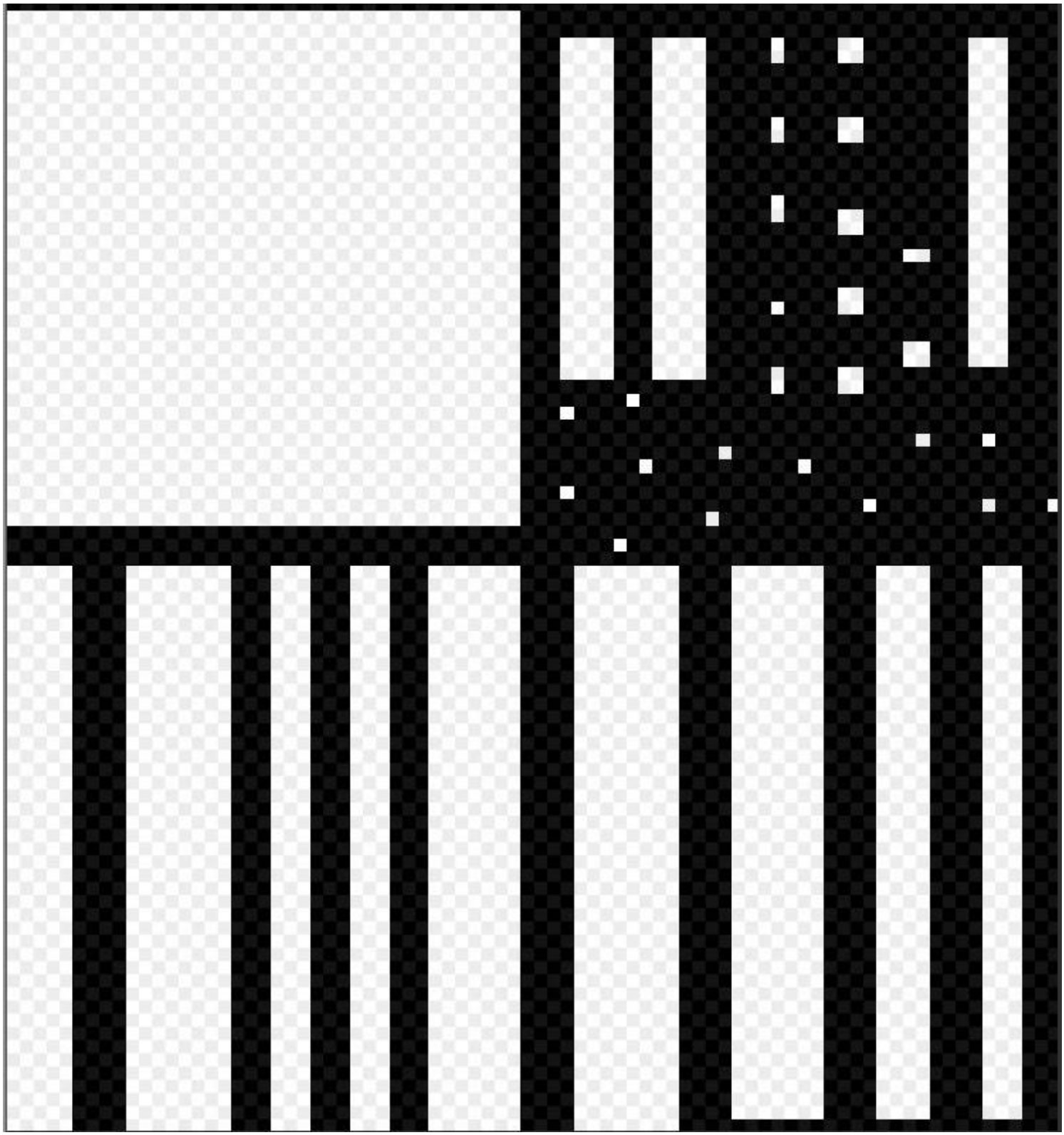
There should be a low difference between the angles in the UV and the angles in the original geometry. See **Angle distortion**, below.

There should be a low difference between the relative scale of triangles in the UV and the relative scale of the triangles in the original geometry), unless you want some areas to have a bigger lightmap resolution. See **Area distortion**, below.

Pack Margin

To allow filtering, the lightmap contains lighting information in texels near the chart border, so always include some margin between charts to avoid light bleeding when applying the lightmap.

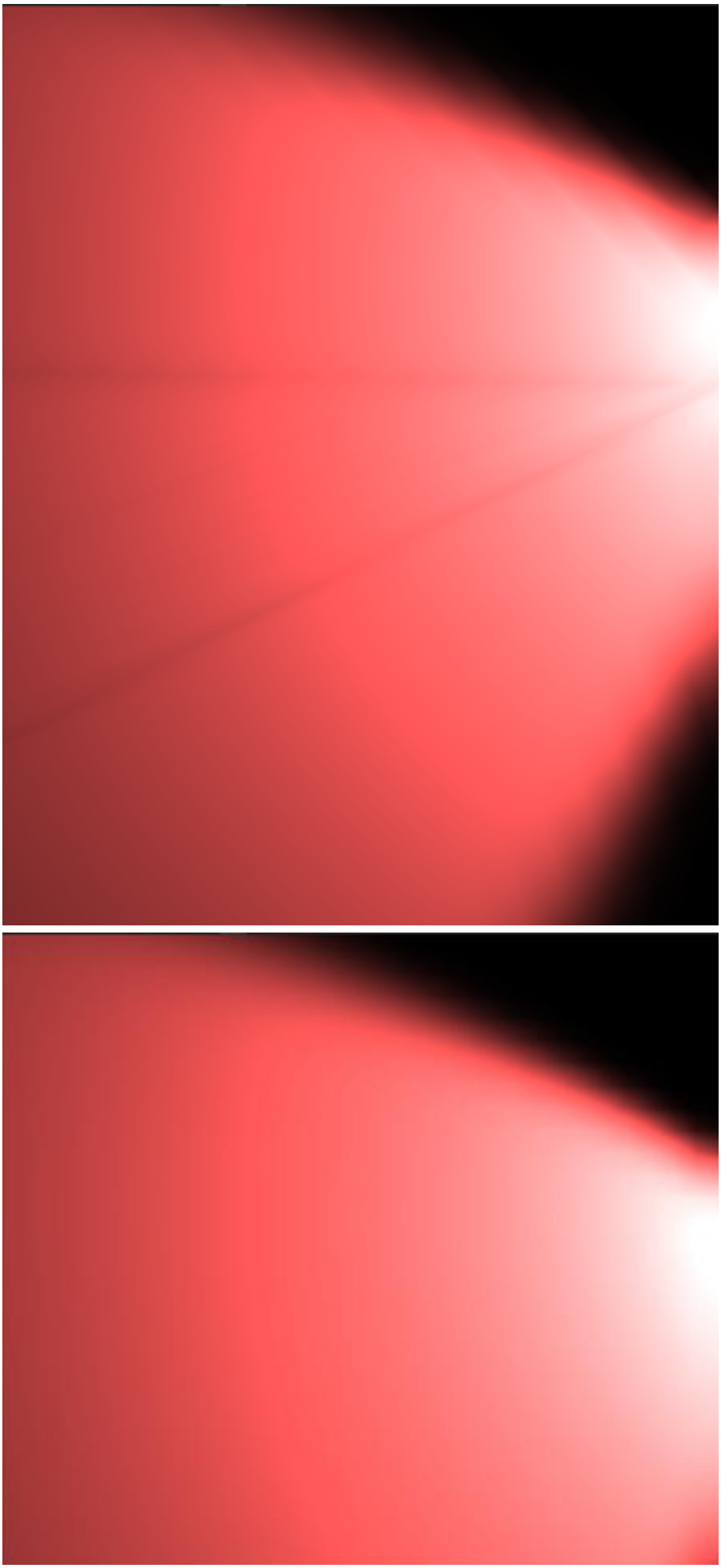
The lightmap resolution defines the texel resolution of your lightmaps. **Lightmappers** dilate some chart texels in the lightmap to avoid black edges, so the UV charts of your Mesh need to be at least two full texels apart from each other to avoid light bleeding. Use the **Pack Margin** setting to ensure you have enough margin between the UV charts of your geometry.



In lightmap UV space, the padding between charts need to be at least two full texels in order to avoid UV overlapping and accidental light bleeding. In this image, the black space represents the space between charts.

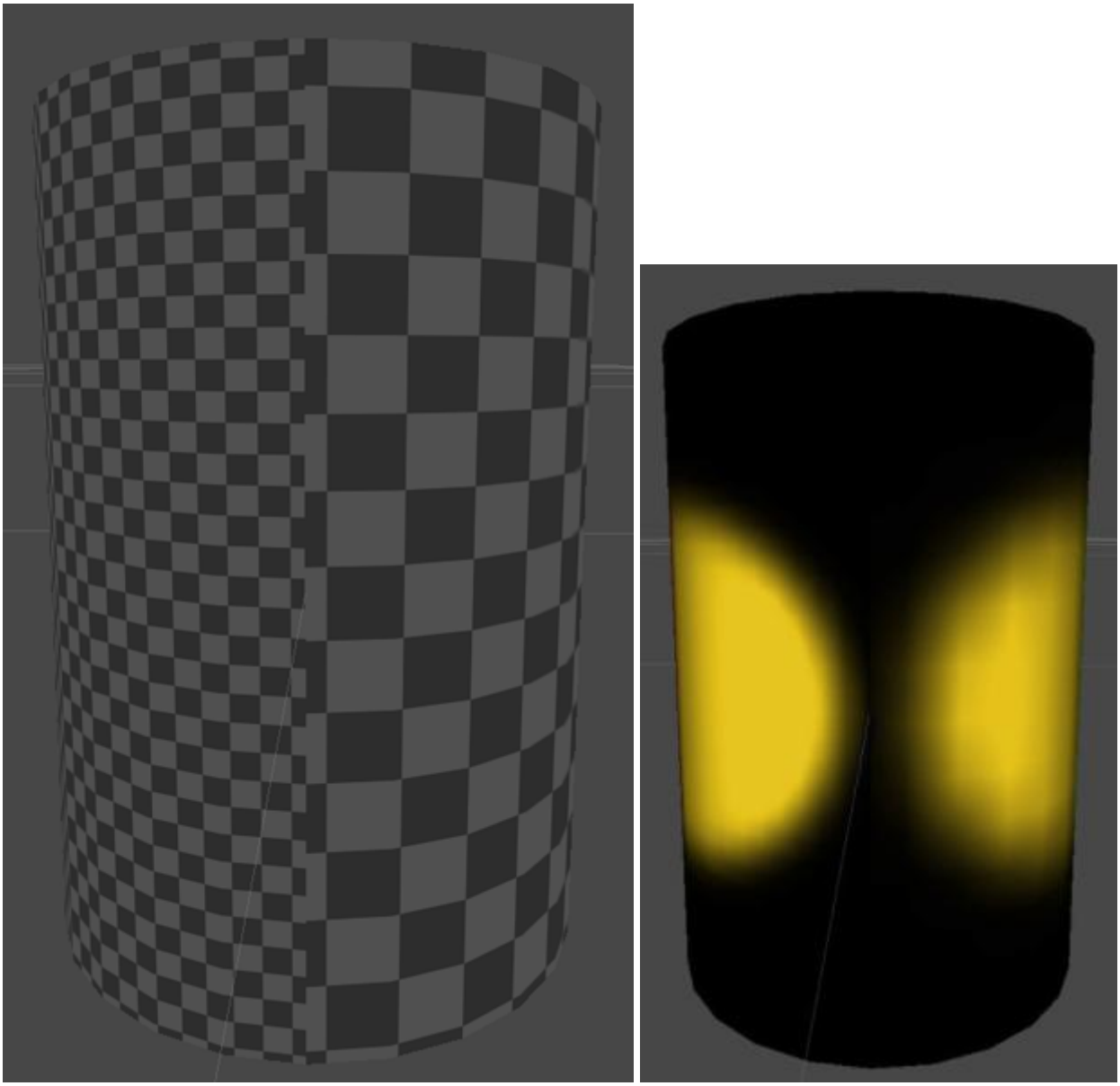
Angle distortion

The following screenshots demonstrate equal resolution, but with different UVs. The first image has a high **Angle Error**, and the result contains unintended artifacts. The second image has the default **Angle Error** (8%). In Meshes with more triangles, angle distortion can significantly distort the shape.



Area distortion

In the image below, two spotlights with the same parameters light the sides of a cylinder. The right-hand side of the cylinder has a higher **Area Error** value, which distorts the triangles and leads to a lower resolution, creating artifacts in the light.



2018–03–28 Page published with limited editorial review

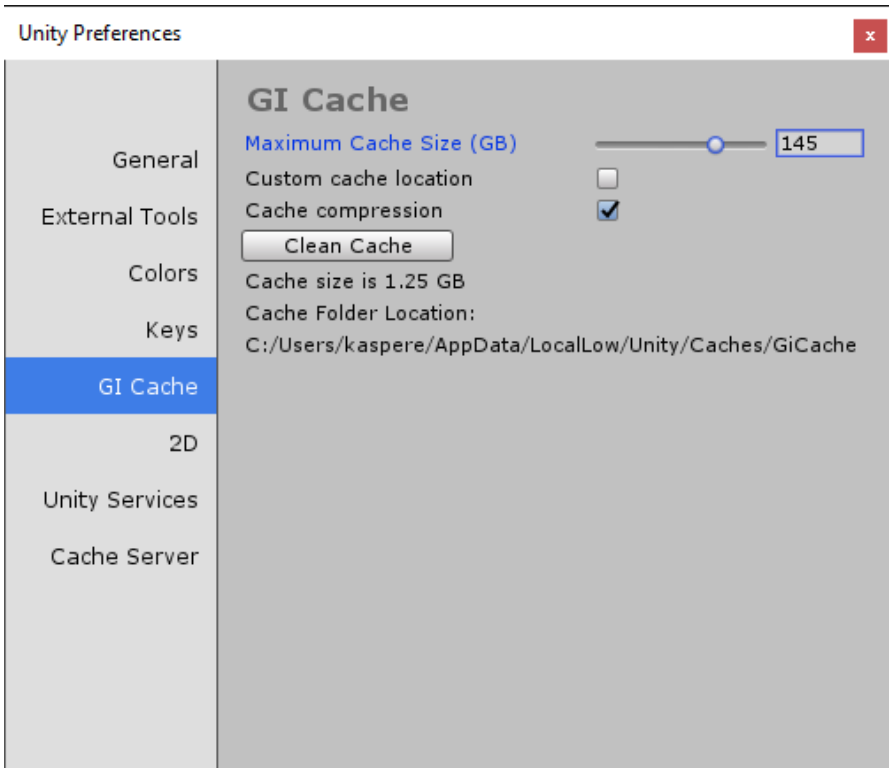
Progressive Lightmapper added in 2018.1

GI cache

Leave feedback

The GI cache is used by the Global Illumination (GI) system to store intermediate files when precomputing real-time GI, and when baking Static **Lightmaps**, **Light Probes** and **Reflection Probes**. The cache is shared between all Unity projects on the computer, so projects with the same content and same version of the lighting system (Enlighten) can share the files and speed up subsequent builds.

Find the settings for the GI cache in **Edit** > **Preferences** > **GI Cache** on Windows, or **Unity** > **Preferences** > **GI Cache** on macOS.



Property:	Function:
Maximum Cache Size (GB)	Use the slider to define the maximum size of the GI cache, in gigabytes. When the GI cache's size grows larger than the size specified in **Preferences** > **GI Cache**, Unity spawns a job to trim the cache. The trim job removes some files so the GI cache does not exceed the size specified. It removes files based on when they were last accessed; it removes those which were last accessed the longest time ago, and keeps those that were most recently used. **Note**: If all the files in the GI cache are currently being used by the current Scene (perhaps because the Scene is very large or the cache size is set too low), increase your cache size. Otherwise, resource-intensive recomputation occurs when baking.
Custom cache location	By default, the GI cache is stored in the *Caches* folder. Tick **Custom cache location** to override the default location and set your own. **Note**: Storing the GI Cache on an SSD drive can speed up baking in cases where the baking process is I/O bound.

Property:	Function:
Cache compression	Tick this checkbox to allow Unity to compress files in the GI cache. The files are LZ4 compressed by default, and the naming scheme is a hash and a file extension. The hashes are computed based on the inputs to the lighting system, so changing any of the following can lead to recomputation of lighting: - Materials (Textures, Albedo, Emission) - Lights - Geometry - Static flags - Light Probe groups - Reflection probes - Lightmap Parameters
Clean Cache	Use the **Clean Cache** button to delete the GI Cache directory. It is not safe to delete the GI Cache directory manually while the Editor is running. This is because the *GiCache* folder is created on Editor startup, and the Editor maintains a collection of hashes that are used to look up the files in the *GiCache* folder. If a file or directory suddenly disappears, the system can't always recover from the failure, and prints an error in the Console. The **Clean Cache** button ensures that the Editor releases all references to the files on disk before they are deleted.

GI cache and lighting

To ensure that the the lighting data loads from the GI cache in a very short amount of time when you reload your Scene, open the Lighting window (menu: **Window** > **Lighting**) and tick the **Auto** checkbox next to the build button. This makes lightmap baking automatic, meaning that the lightmap data is stored in the GI cache.

In the Lighting window, you can clear the baked data in a **Scene** (untick the **Auto** checkbox, click the **Build** button drop-down and select **Clear Baked Data**). This does not clear the GI Cache, because this would increase bake time afterwards.

You can share the *GiCache* folder among different machines. This can make your lighting build faster, because the files are downloaded from the *GiCache* folder instead of computed locally. Note that the build process isn't optimized for slow network-attached storage (NAS), so test if your bake times are severely affected before moving the cache to NAS.

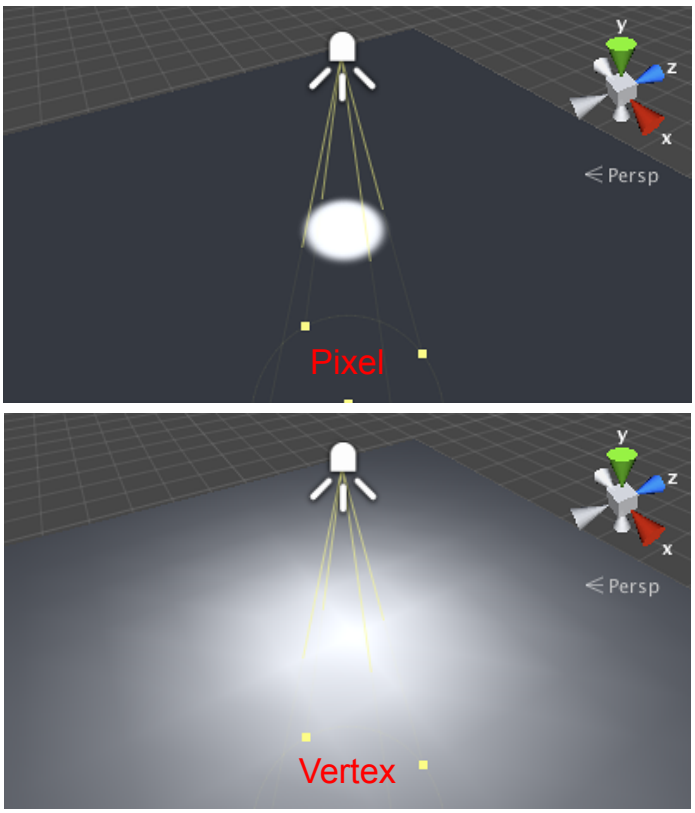
Light troubleshooting and performance

Leave feedback

Lights can be rendered using either of two methods:

Vertex lighting calculates the illumination only at the vertices of meshes and interpolates the vertex values over the rest of the surface. Some lighting effects are not supported by vertex lighting but it is the cheaper of the two methods in terms of processing overhead. Also, this may be the only method available on older graphics cards.

Pixel lighting is calculated separately at every screen **pixel**. While slower to render, **pixel** lighting does allow some effects that are not possible with vertex lighting. Normal-mapping, light cookies and realtime shadows are only rendered for **pixel** lights. Additionally, spotlight shapes and point light highlights look much better when rendered in **pixel** mode.



Comparison of a spotlight rendered in pixel vs vertex mode

Lights have a big impact on **rendering** speed, so lighting quality must be traded off against frame rate. Since pixel lights have a much higher **rendering** overhead than vertex lights, Unity will only render the brightest lights at per-pixel quality and render the rest as vertex lights. The maximum number of pixel lights can be set in the Quality Settings for standalone build targets.

You can favour a light to be rendered as a pixel light using its **Render Mode** property. A light with the mode set to **Important** will be given higher priority when deciding whether or not to render it as a pixel light. With the mode set to **Auto** (the default), Unity will classify the light automatically based on how much a given object is affected by the light. The lights that are rendered as pixel lights are determined on an object-by-object basis.

See the page about Optimizing Graphics Performance for further information.

Lighting window statistics

The bottom of the Lighting window displays statistics showing important metrics with regard to run time performance. See documentation on the Lighting window for more details.

Shadow performance

Realtime shadows have quite a high rendering overhead, so you should use them sparingly. Any objects that might cast shadows must first be rendered into the shadow map and then that map will be used to render objects that might receive shadows. Enabling shadows has an even bigger impact on performance than the pixel/vertex trade-off mentioned above.

Soft shadows have a greater rendering overhead than hard shadows but this only affects the GPU and does not cause much extra CPU work.

The Quality Settings include a **Shadow Distance** value. Objects that are beyond this distance from the **camera** will be rendered with no shadows at all. Since the shadows on distant objects will not usually be noticed anyway, this can be a useful optimisation to reduce the number of shadows that must be rendered.

A particular issue with directional lights is that a single light can potentially illuminate the whole of a **scene**. This means that the shadow map will often cover a large portion of the **scene** at once and this makes the shadows susceptible to a problem known as "perspective aliasing". Simply put, perspective aliasing means that shadow map pixels seen close to the camera look enlarged and "chunky" compared to those farther away. Although you can just increase the shadow map resolution to reduce this effect, the result is that rendering resources are wasted for distant areas whose shadow map looked fine at the lower resolution.

A good solution to the problem is therefore to use separate shadow maps that decrease in resolution as the distance from camera increases. These separate maps are known as **cascades**. From the Quality Settings, you can choose zero, two or four cascades; Unity will calculate the positioning of the cascades within the camera's frustum. Note that cascades are only enabled for directional lights. See directional light shadows page for details.

How the size of a shadow map is calculated

The first step in calculating the size of the map is to determine the area of the screen view that the light can illuminate. For directional lights, the whole screen can be illuminated but for spot lights and point lights, the area is the onscreen projection of the shape of the light's extent (a sphere for point lights or a cone for spot lights). The projected shape has a certain width and height in pixels on the screen; the larger of those two values is then taken as the light's "pixel size".

When the shadow map resolution is set to **High** (from the Quality Settings) the shadow map's size is calculated as follows:

Directional lights: NextPowerOfTwo (pixelSize * 1.9), up to a maximum of 2048.
Spot lights: NextPowerOfTwo (pixelSize), up to a maximum of 1024.
Point lights: NextPowerOfTwo (pixelSize * 0.5), up to a maximum of 512.
If the graphics card has 512MB or more video memory, the upper shadow map limits are increased to 4096 for directional lights, 2048 for spot lights and 1024 for point lights.

At *Medium* shadow resolution, the shadow map size is half the value for **High** resolution and for *Low*, it is a quarter of the size.

Point lights have a lower limit on size than the other types is because they use **cubemaps** for shadows. That means that six cubemap faces at this resolution must be kept in video memory at once. They are also quite expensive to render, as potential shadow casters might need to be rendered into all six cubemap faces.

Troubleshooting shadows

If you find that one or more objects are not casting shadows then you should check the following points:

Old graphics hardware may not support shadows. See below for a list of minimal hardware specs that can handle shadows.

Shadows can be disabled in the Quality Settings. Make sure that you have the correct quality level enabled and that shadows are switched on for that setting.

All Mesh Renderers in the scene must be set up with their *Receive Shadows* and *Cast Shadows* correctly set. Both are enabled by default but check that they haven't been disabled unintentionally.

Only opaque objects cast and receive shadows so objects using the built-in Transparent or Particle shaders will neither cast nor receive. Generally, you can use the Transparent Cutout shaders instead for objects with "gaps" such as fences, vegetation, **etc**. Custom Shaders must be pixel-lit and use the Geometry render queue.

Objects using **VertexLit** shaders can't receive shadows but they can cast them.

With the Forward rendering path, some shaders allow only the brightest directional light to cast shadows (in particular, this happens with Unity's legacy built-in shaders from 4.x versions). If you want to have more than one shadow-casting light then you should use the Deferred Shading **rendering path** instead. You can enabled your own shaders to support "full shadows" by using the `fullforwardshadows` surface shader directive.

Hardware support for shadows

Built-in shadows work on almost all devices supported by Unity. The following cards are supported on each platform:

PC (Windows/Mac/Linux)

Generally all GPUs support shadows. Exceptions might occur in some really old GPUs (for example, Intel GPUs made in 2005).

Mobile

iPhone 4 does not support shadows. All later models starting with iPhone 4S and iPad 2 support shadows.
Android: Requires Android 4.0 or later, and `GL_OES_depth_texture` support. Most notably, some Android Tegra 2/3-based Android devices do not have this, so they don't support shadows.
Windows Phone: Shadows are only supported on DX11-class GPUs (Adreno 4xx/5xx).

Consoles

All consoles support shadows.

2017–06–08 Page published with limited editorial review

Lighting window statistics added in 5.6

Related topics

Leave feedback

Unity's lighting system is affected by and can affect many of its other effects and systems.

Quality settings

The Quality settings window has many settings that affects lighting and shadows.

Player settings

The Unity Player settings window allows you to choose the **rendering path** and color space.

Camera inspector

The Camera inspector allows you to override Unity **Player settings** for the **rendering** path per **camera**. **HDR** can also be activated here.

Rendering paths

Unity supports a number of rendering techniques, or 'paths'. An important early decision which needs to be made when starting a project is which path to use. Unity's default is '**Forward Rendering**".

In Forward Rendering, each object is rendered in a 'pass' for each light that affects it. Therefore each object might be rendered multiple times depending upon how many lights are within range.

The advantages of this approach is that it can be very fast - meaning hardware requirements are lower than alternatives. Additionally, Forward Rendering offers us a wide range of custom 'shading models' and can handle transparency quickly. It also allows for the use of hardware techniques like 'multi-sample anti-aliasing' (MSAA) which are not available in other alternatives, such as Deferred Rendering which can have a great impact on image quality.

However, a significant disadvantage of the forward path is that we have to pay a render cost on a per-light basis. That is to say, the more lights affecting each object, the slower rendering performance will become. For some game types, with lots of lights, this may therefore be prohibitive. However if it is possible to manage light counts in your game, Forward Rendering can actually be a very fast solution.

In 'Deferred' rendering, on the other hand, we defer the shading and blending of light information until after a first pass over the screen where positions, normals, and materials for each surface are rendered to a 'geometry buffer' (G-buffer) as a series of screen-space textures. We then composite these results together with the lighting pass. This approach has the principle advantage that the render cost of lighting is proportional to the number of **pixels** that the light illuminates, instead of the number of lights themselves. As a result you are no longer bound by the number of lights you wish to render on screen, and for some games this is a critical advantage.

Deferred Rendering gives highly predictable performance characteristics, but generally requires more powerful hardware. It is also not supported by certain mobile hardware.

For more information on the Deferred, Forward and the other available rendering paths, please see the main documentation page.

High dynamic range

High dynamic range rendering allows you to simulate a much wider range of colours than has been traditionally available. In turn, this usually means you need to choose a range of brightnesses to display on the screen. In this way it is possible to simulate great differences in brightness between, say, outdoor lighting in our **scenes** and shaded areas. We can also create effects like 'blooms' or glows by applying effects to these bright colors in your scene. Special effects like these can add realism to particles or other visible light sources.

For more information about HDR, please see the relevant manual page.

Tonemapping

Tonemapping is part of the color grading post-processing effect that is required to describe how to map colours from HDR to what you can see on the screen. Please see the Color Grading effect for more information.

Reflection

While not explicitly a lighting effect, reflections are important for realistically displaying materials that reflect light, such as shiny metals or glass. Modern shading techniques, including Unity's **Standard Shader**, integrate reflection into a material's properties.

For more information, please refer to our section on Reflections.

Linear color space

In addition to selecting a rendering path, it's important to choose a 'Color Space' before lighting your project. Color Space determines the maths used by Unity when mixing colors in lighting calculations or reading values from textures. This can have a drastic effect on the realism of your game, but in many cases the decision over which Color Space to use will likely be forced by the hardware limitations of your target platform.

The preferred Color Space for realistic rendering is Linear.

A significant advantage of using Linear space is that the colors supplied to **shaders** within your scene will brighten linearly as light intensities increase. With the alternative, 'Gamma' Color Space, brightness will quickly begin to turn to white as values go up, which is detrimental to image quality.

For further information please see Linear rendering.

Linear rendering overview

Leave feedback

The Unity Editor allows you to work in traditional gamma color space as well as linear color space. While gamma color space is the historically standard format, linear color space **rendering** gives more precise results.

For further reading, see documentation on:

Linear or gamma workflow for information on selecting to work in linear or gamma color space.
Gamma Textures with linear rendering for information on gamma Textures in a linear workflow.
Linear Textures for information on working with linear Textures.

Linear and gamma color space

The human eye doesn't have a linear response to light intensity. We see some brightnesses of light more easily than others - a gradient that proceeds in a linear fashion from black to white would not look like a linear gradient to our eyes.

Left: A linear gradient. Right: How our eyes perceive that gradient. Note where the borders (which are exactly mid-grey) merge with the gradient in each case

For historical reasons, monitors and displays have the same characteristic. Sending a monitor a linear signal results in something that looks like the gradient to the right in the illustration above, and simply looks wrong to our eyes. To compensate for this, a corrected signal is sent to make sure the monitor shows an image that looks natural. This correction is known as gamma correction.

The reason both gamma and linear color spaces exist is because lighting calculations should be done in linear space in order to be mathematically correct, but the result should be presented in gamma space to look correct to our eyes.

When calculating lighting on older hardware restricted to 8 bits per channel for the framebuffer format, using a gamma curve provides more precision in the human-perceivable range. More bits are used in the range where the human eye is the most sensitive.

Even though monitors today are digital, they still take a gamma-encoded signal as input. Image files and video files are explicitly encoded to be in gamma space (meaning they carry gamma-encoded values, not linear intensities). This is the standard; everything is in gamma space.

The accepted standard for gamma space is called sRGB (see Wikipedia). This standard defines a mapping to linear space that allows our eyes to make the most of the 8 bits per channel of precision. Below is a diagram of this mapping.

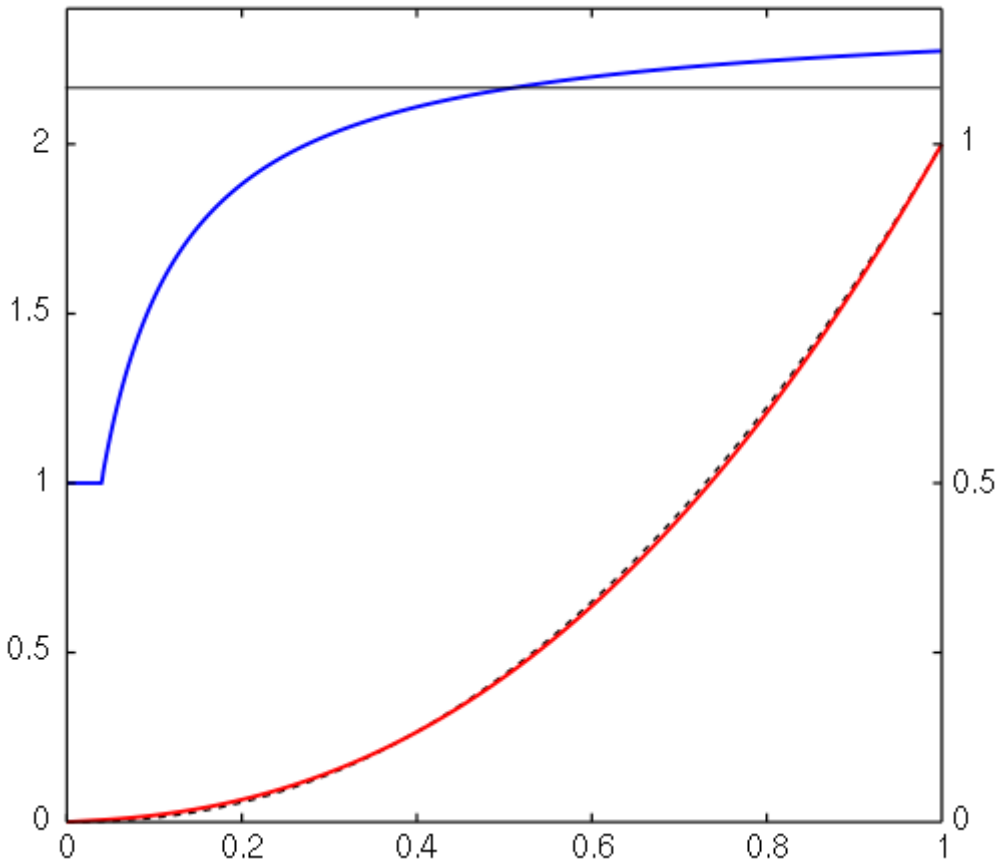


Image courtesy of Wikimedia. License: public domain

Linear rendering refers to the process of rendering a **Scene** with all inputs linear - that is to say, not gamma corrected for viewing with human eyes or for output to a display.

Linear or gamma workflow

Leave feedback

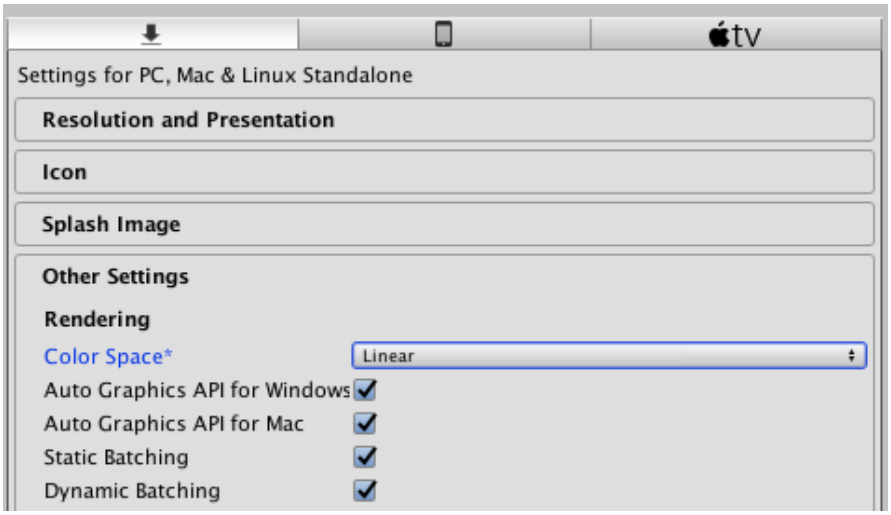
The Unity Editor offers both linear and gamma workflows. The linear workflow has a color space crossover where Textures that were authored in gamma color space can be correctly and precisely rendered in linear color space. See documentation on Linear rendering overview for more information about gamma and linear color space.

For further reading, see documentation on:

Linear rendering overview for background information on linear and gamma color space.
Gamma Textures with linear rendering for information on gamma Textures in a linear workflow.
Linear Textures for information on working with linear Textures.

Textures tend to be saved in gamma color space, while Shaders expect linear color space. As such, when Textures are sampled in Shaders, the gamma-based values lead to inaccurate results. To overcome this, you can set Unity to use an RGB sampler to cross over from gamma to linear sampling. This ensures a linear workflow with all inputs and outputs of a **Shader** in the correct color space, resulting in a correct outcome.

To specify a gamma or linear workflow, go to **Edit** > **Project Settings** > **Player** and open **Player Settings**. Go to **Other Settings** > **Rendering** and change the **Color Space** to **Linear** or **Gamma**, depending on your preference.



The Player Settings window showing the **Color Space** setting

Gamma color space workflow

While a linear workflow ensures more precise rendering, sometimes you may want a gamma workflow (for example, on some platforms the hardware only supports the gamma format).

To do this, set **Color Space** to **Gamma** in the Player Settings window (menu: **Edit** > **Project Settings** > **Player**). With this option selected, the rendering pipeline uses all colors and textures in the gamma color space in which they are stored - textures do not have gamma correction removed from them when they are used in a Shader.

Note that you can choose to bypass sRGB sampling in **Color Space: Gamma** mode by unchecking the **sRGB (Color Texture)** checkbox in the Inspector window for the Texture.

Note: Even though these values are in gamma space, all the Unity Editor's Shader calculations still treat their inputs as if they were in linear space. To ensure an acceptable final result, the Editor makes an adjustment to deal with the mismatched formats when it writes the Shader outputs to a framebuffer and does not apply gamma correction to the final result.

Linear color space workflow

Working in linear color space gives more accurate rendering than working in gamma color space.

To do this, set **Color Space** to **Linear** in the Player Settings window (menu: **Edit** > **Project Settings** > **Player**).

You can work in linear color space if your Textures were created in linear or gamma color space. Gamma color space Texture inputs to the linear color space Shader program are supplied to the Shader with gamma correction removed from them.

Linear Textures

Selecting **Color Space: Linear** assumes your Textures are in gamma color space. Unity uses your GPU's sRGB sampler by default to crossover from gamma to linear color space. If your Textures are authored in linear color space, you need to bypass the sRGB sampling. See documentation on Working with linear Textures for more information.

Gamma Textures

Crossing over from gamma color space to linear color space requires some tweaking. See documentation on Gamma Textures with linear rendering for more information.

Notes

For colors, this conversion is applied implicitly, because the Unity Editor already converts the values to floating point before passing them to the GPU as constants. When sampling Textures, the GPU automatically removes the gamma correction, converting the result to linear space.

These inputs are then passed to the Shader, with lighting calculations taking place in linear space as they normally do. When writing the resulting value to a framebuffer, it is either gamma-corrected straight away or left in linear space for later gamma correction - this depends on the current rendering configuration. For example, in high dynamic range (HDR), rendering results are left in linear space and gamma corrected later.

Differences between linear and gamma color space

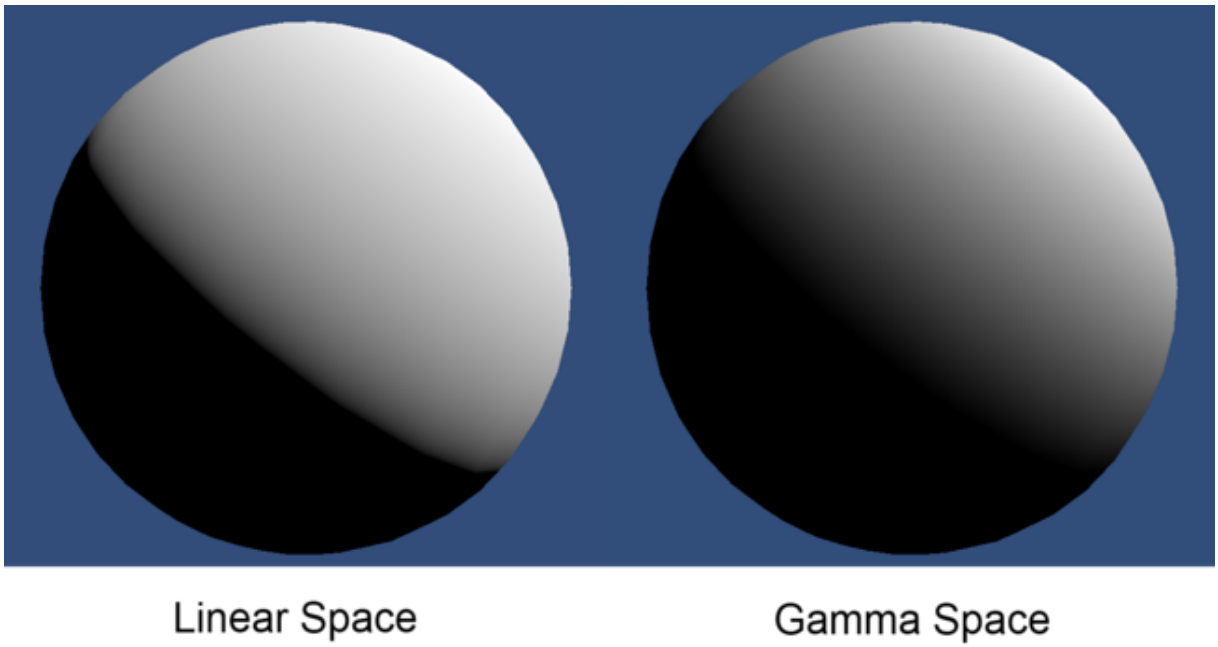
When using linear rendering, input values to the lighting equations are different to those in gamma space. This means differing results depending on the color space. For example, light striking surfaces has differing response curves, and Image Effects behave differently.

Light fall-off

The fall-off from distance and normal-based lighting differs in two ways:

When rendering in linear mode, the additional gamma correction that is performed makes a light's radius appear larger.

Lighting edges also appear more clearly. This more correctly models lighting intensity fall-off on surfaces.

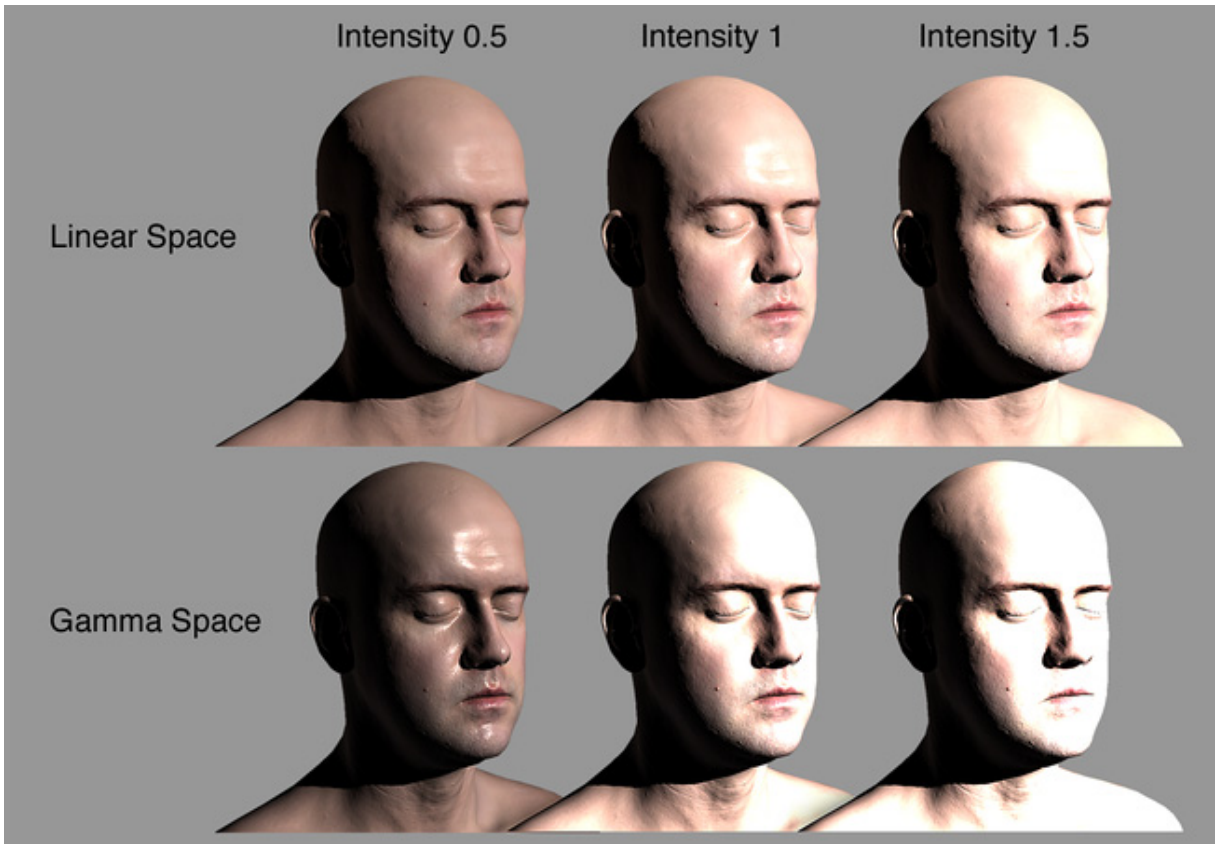


Left: Lighting a sphere in linear space. Right: Lighting a sphere in gamma space

Linear intensity response

When you are using gamma rendering, the colors and Textures that are supplied to a Shader already have gamma correction applied to them. When they are used in a Shader, the colors of high luminance are actually brighter than they should be compared to linear lighting. This means that as light intensity increases, the surface gets brighter in a nonlinear way. This leads to lighting that can be too bright in many places. It can also give models and **scenes** a washed-out feel. When you are using linear rendering, the response from the surface remains linear as the light intensity increases. This leads to much more realistic surface shading and a much nicer color response from the surface.

The Infinite 3D Head Scan image below demonstrates different light intensities on a human head model under linear lighting and gamma lighting.



Infinite 3D Head Scan by Lee Perry-Smith, licensed under a Creative Commons Attribution 3.0 Unported License (available from www.ir-ltd.net)

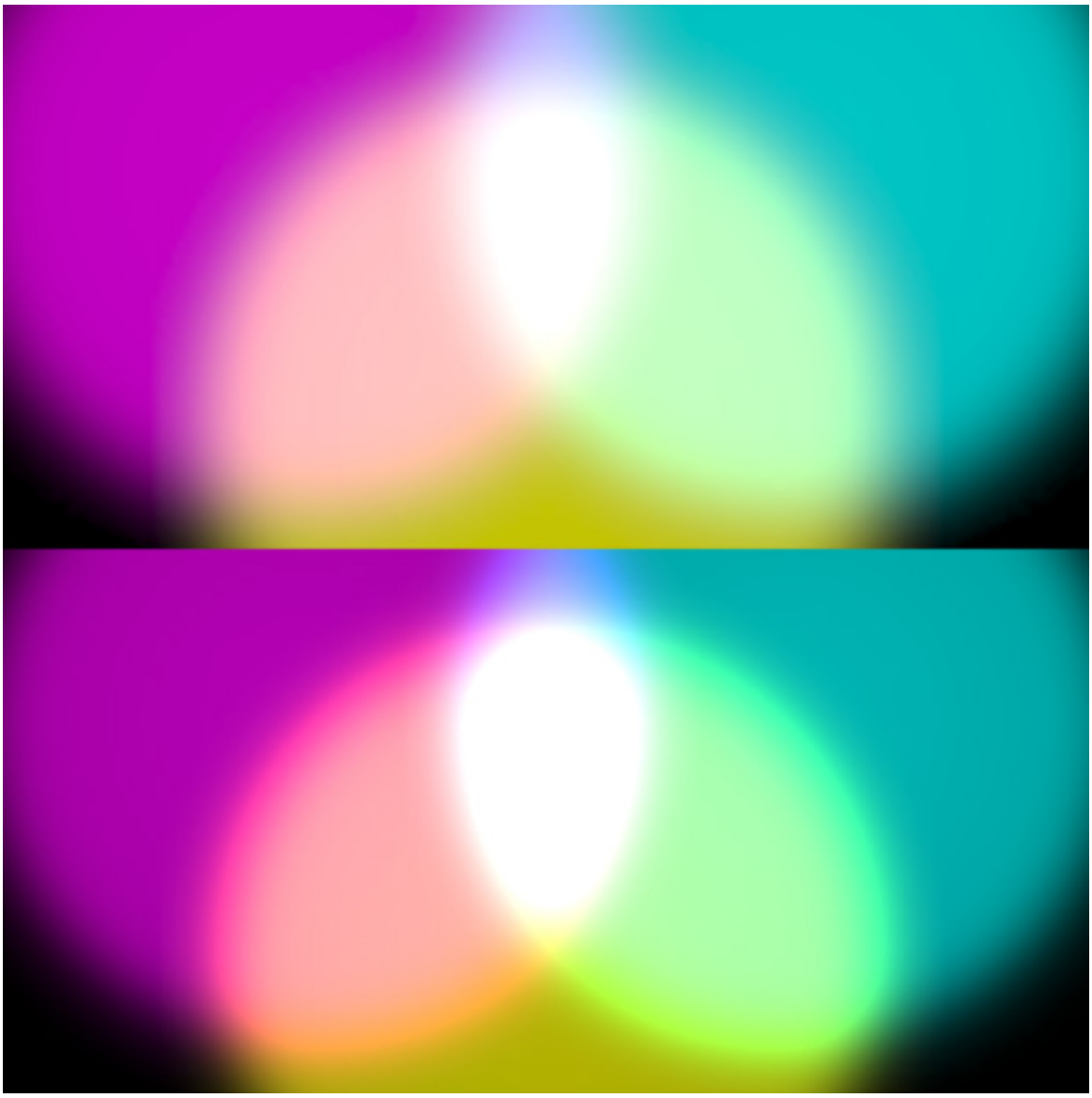
Linear and gamma blending

When blending into a framebuffer, the blending occurs in the color space of the framebuffer.

When you use gamma space rendering, nonlinear colors get blended together. This is not the mathematically correct way to blend colors, and can give unexpected results, but it is the only way to do a blend on some graphics hardware.

When you use linear space rendering, blending occurs in linear color space: This is mathematically correct and gives precise results.

The image below demonstrates the different types of blending:



Top: Blending in linear color space produces expected blending results
Bottom: Blending in gamma color space results in over-saturated and overly-bright blends

Gamma Textures with linear rendering

Leave feedback

The Unity Editor allows you to work with traditional gamma color space as well as linear color space. You can work in linear colour space even if your Textures are in gamma color space.

For further reading, see documentation on:

Linear rendering overview for background information on linear and gamma color space.
Linear or gamma workflow for information on selecting to work in linear or gamma color space.
Linear Textures for information on working with linear Textures.
Note: If your Textures are in linear color space, you need to disable sRGB sampling. See documentation on Linear Textures for more information.

Linear **rendering** gives a different look to the rendered **Scene**. When you have authored a project to look good when **rendering** in gamma space, it is unlikely to look great when you change to linear **rendering**. Because of this, if you move to linear **rendering** from gamma **rendering** it may take some time to tweak the project so that it looks as good as before. However, the switch ultimately enables more consistent and realistic **rendering** and so may be worth the time spent on it. You are likely to have to tweak Textures, Materials and Lights.

Lightmapping

The lighting calculations in the **lightmapper** are always done in linear space (see documentation on the Lighting Window for more information). The lightmaps are always stored in gamma space. This means that the lightmap textures are identical no matter whether you're in gamma or linear color space.

When you are in linear color space, the texture sample gets converted from gamma to linear space when sampling the texture. When you're in gamma color space, no conversion is needed. Therefore, when you change the color space setting, you must rebake lightmaps: This happens automatically when Unity's lighting is set to auto bake (which is the default).

Importing lightmaps

The data in lightmap EXR files created by Unity is in linear space. It gets converted to gamma space during import. When bringing in lightmaps from an external lightmapper, mark the lightmaps as **Texture Type**: **Lightmap** in the Texture Importer. This setting makes sure sRGB sampling is bypassed on import.

Linear supported platforms

Linear rendering is not supported on all platforms. The build targets that support the feature are:

Windows, Mac OS X and Linux (Standalone)
Xbox One
PlayStation 4
Android
iOS
WebGL
There is no fallback to gamma when linear rendering is not supported by the device. In this situation, the Player quits. You can check the active color space from a script by looking at QualitySettings.activeColorSpace.

On Android, linear rendering requires at least OpenGL ES 3.0 graphics API and Android 4.3.

On **iOS**, linear rendering requires the Metal graphics API.

On **WebGL**, linear rendering requires at least **WebGL** 2.0 graphics API.

Until the minimum requirements are satisfied, the Editor prevents you from building a Player and shows a notification. This is to avoid games that would render incorrectly on user devices being deployed to digital stores.

The Unity Editor prevents building a Player for games that would render incorrectly

Linear color space and HDR

When using **HDR**, rendering is performed in linear space into floating point buffers. These buffers have enough precision not to require conversion to and from gamma space whenever the buffer is accessed. This means that when rendering in linear mode, the framebuffers you use store the colors in linear space. Therefore, all blending and post process effects are implicitly performed in linear space. When the final backbuffer is written to, gamma correction is applied.

Linear color space and non-HDR

When linear color space is enabled and HDR is not enabled, a special framebuffer type is used that supports sRGB read and sRGB write (convert from gamma to linear when reading, convert from linear to gamma when writing). When this framebuffer is used for blending or it is bound as a Texture, the values are converted to linear space before being used. When these buffers are written to, the value that is being written is converted from linear space to gamma space. If you are rendering in linear mode and non-HDR mode, all post-process effects have their source and target buffers created with sRGB read and write enabled so that post-processing and post-process blending occur in linear space.

2017–06–19 Page amended with no editorial review

Linear rendering for WebGL added in 2017.2

Working with linear Textures

Leave feedback

sRGB sampling allows the Unity Editor to render **Shaders** in linear color space when Textures are in gamma color space. When you select to work in linear color space, the Editor defaults to using sRGB sampling. If your Textures are in linear color space, you need to work in linear color space and disable sRGB sampling for each Texture. To learn how to do this, see Disabling sRGB sampling, below.

For further reading, see documentation on:

Linear rendering overview for background information on linear and gamma color space.
Linear or gamma workflow for information on selecting to work in linear or gamma color space.
Gamma Textures with linear rendering for information on gamma Textures in a linear workflow.

Legacy GUI

Rendering of elements of the Legacy GUI System is always done in gamma space. This means that for the legacy GUI system, Textures with their **Texture Type** set to **Editor GUI and Legacy GUI** do not have their gamma removed on import.

Linear authored Textures

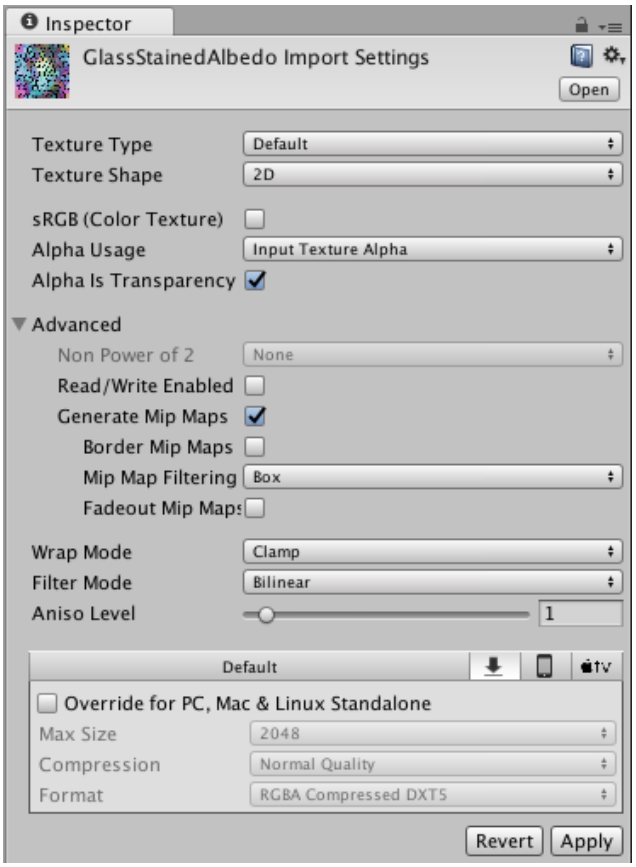
It is also important that lookup Textures, masks, and other textures with RGB values that mean something specific and have no gamma correction applied to them bypass sRGB sampling. This prevents values from the sampled Texture having non-existent gamma correction removed before they are used in the Shader, with calculations made with the same value as is stored on disk. Unity assumes that GUI textures and normal map textures are authored in a linear space.

Disabling sRGB sampling

To ensure a Texture is imported as a linear color space image, in the Inspector window for the Texture:

Select the appropriate **Texture Type** for the Texture's intended use.

Uncheck **sRGB (Color Texture)** if it is shown.



The Inspector window for a Texture. Note the setting for **sRGB (Color Texture)** is unchecked. This ensures the Texture is imported as a linear color space image

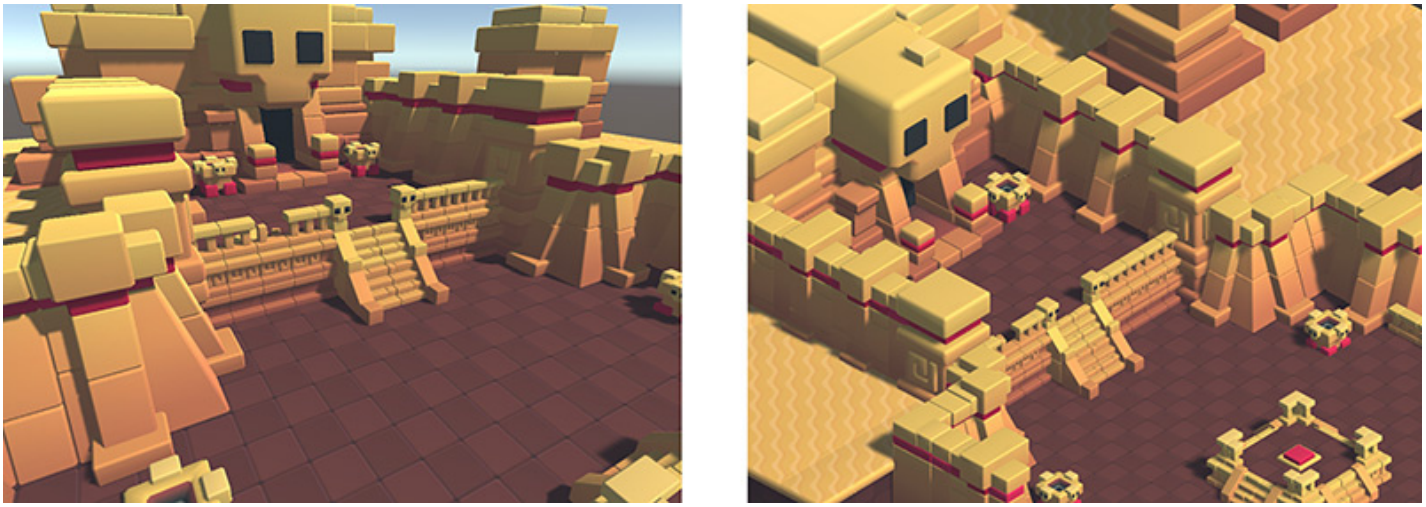
Cameras

Leave feedback

A Unity **scene** is created by arranging and moving objects in a three-dimensional space. Since the viewer's screen is two-dimensional, there needs to be a way to capture a view and "flatten" it for display. This is accomplished using **Cameras**.

A camera is an object that defines a view in scene space. The object's position defines the viewpoint, while the forward (Z) and upward (Y) axes of the object define the view direction and the top of the screen, respectively. The Camera component also defines the size and shape of the region that falls within the view. With these parameters set up, the camera can display what it currently "sees" to the screen. As the camera object moves and rotates, the displayed view will also move and rotate accordingly.

Perspective and orthographic cameras



The same scene shown in perspective mode (left) and orthographic mode (right)

A camera in the real world, or indeed a human eye, sees the world in a way that makes objects look smaller the farther they are from the point of view. This well-known *perspective* effect is widely used in art and computer graphics and is important for creating a realistic scene. Naturally, Unity supports perspective cameras, but for some purposes, you want to render the view without this effect. For example, you might want to create a map or information display that is not supposed to appear exactly like a real-world object. A camera that does not diminish the size of objects with distance is referred to as *orthographic* and Unity cameras also have an option for this. The perspective and orthographic modes of viewing a scene are known as camera *projections*. *(scene above from BITGEM)*

The shape of the viewed region

Both perspective and orthographic cameras have a limit on how far they can "see" from their current position. The limit is defined by a plane that is perpendicular to the camera's forward (Z) direction. This is known as the *far clipping plane* since objects at a greater distance from the camera are "clipped" (ie, excluded from rendering). There is also a corresponding *near clipping plane* close to the camera - the viewable range of distance is that between the two planes.

Without perspective, objects appear the same size regardless of their distance. This means that the viewing volume of an orthographic camera is defined by a rectangular box extending between the two **clipping planes**.

When perspective is used, objects appear to diminish in size as the distance from camera increases. This means that the width and height of the viewable part of the scene grows with increasing distance. The viewing volume of a perspective camera, then, is not a box but a pyramidal shape with the apex at the camera's position and the base at the **far clipping plane**. The shape is not exactly a pyramid, however, because the top is cut off by the **near clipping plane**; this kind of truncated pyramid shape is known as a *frustum*. Since its height is not constant, the frustum is defined by the ratio of its width to its height (known as the *aspect ratio*) and the angle between the top and bottom at the apex (known as the *field of view* of *FOV*). See the page about understanding the view frustum for a more detailed explanation.

The background to the camera view

For indoor scenes, the camera may always be completely inside some object representing the interior of a building, cave or other structure. When the action takes place outdoors, however, there will be many empty areas in between objects that are filled with nothing at all; these background areas typically represent the sky, space or the murky depths of an underwater scene.

A camera can't leave the background completely undecided and so it must fill in the empty space with something. The simplest option is to clear the background to a flat color before rendering the scene on top of it. You can set this color using the camera's *Background* property, either from the **inspector** or from a script. A more sophisticated approach that works well with outdoor scenes is to use a Skybox. As its name suggests, a skybox behaves like a "box" lined with images of a sky. The camera is effectively placed at the center of this box and can see the sky from all directions. The camera sees a different area of sky as it rotates but it never moves from the center (so the camera cannot get "closer" to the sky). The skybox is rendered behind all objects in the scene and so it represents a view at infinite distance. The most common usage is to represent the sky in a standard outdoor scene but the box actually surrounds the camera completely, even underneath. This means that you can use a skybox to represent parts of the scene (eg, rolling plains that stretch beyond the horizon) or the all-round view of a scene in space or underwater.

You can add a skybox to a scene simply by setting the *Skybox* property in the Lighting window (menu: **Window** > **Rendering** > **Lighting Settings**). See this page for further details on how to create your own skybox.

Using more than one camera

Leave feedback

When created, a Unity **scene** contains just a single **camera** and this is all you need for most situations. However, you can have as many cameras in a **scene** as you like and their views can be combined in different ways, as described below.

Switching cameras

By default, a camera renders its view to cover the whole screen and so only one camera view can be seen at a time (the visible camera is the one that has the highest value for its *depth* property). By disabling one camera and enabling another from a script, you can "cut" from one camera to another to give different views of a scene. You might do this, for example, to switch between an overhead map view and a first-person view.

```
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public Camera firstPersonCamera;
    public Camera overheadCamera;

    public void ShowOverheadView() {
        firstPersonCamera.enabled = false;
        overheadCamera.enabled = true;
    }

    public void ShowFirstPersonView() {
        firstPersonCamera.enabled = true;
        overheadCamera.enabled = false;
    }
}
```

Example C# script

```
var firstPersonCamera: Camera;
var overheadCamera: Camera;

function ShowOverheadView() {
    firstPersonCamera.enabled = false;
    overheadCamera.enabled = true;
}

function ShowFirstPersonView() {
    firstPersonCamera.enabled = true;
```

```
    overheadCamera.enabled = false;
}
```

Example JS script

Rendering a small camera view inside a larger one

Usually, you want at least one camera view covering the whole screen (the default setting) but it is often useful to show another view inside a small area of the screen. For example, you might show a rear view mirror in a driving game or show an overhead mini-map in the corner of the screen while the main view is first-person. You can set the size of a camera's onscreen rectangle using its *Viewport Rect* property.

The coordinates of the **viewport** rectangle are "normalized" with respect to the screen. The bottom and left edges are at the 0.0 coordinate, while the top and right edges are at 1.0. A coordinate value of 0.5 is halfway across. In addition to the **viewport** size, you should also set the *depth* property of the camera with the smaller view to a higher value than the background camera. The exact value does not matter but the rule is that a camera with a higher depth value is rendered over one with a lower value.

Camera Tricks

Leave feedback

It is useful to understand how the **camera** works when designing certain visual effects or interactions with objects in the **scene**. This section explains the nature of the **camera**'s view and how it can be used to enhance gameplay.

Understanding the View Frustum

Leave feedback

The word **frustum** refers to a solid shape that looks like a pyramid with the top cut off parallel to the base. This is the shape of the region that can be seen and rendered by a perspective **camera**. The following thought experiment should help to explain why this is the case.

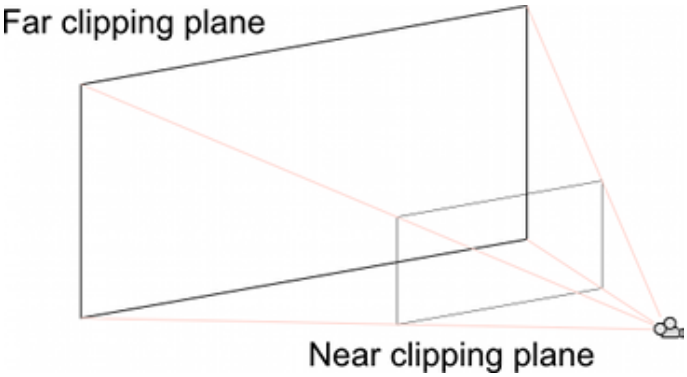
Imagine holding a straight rod (a broom handle or a pencil, for example) end-on to a camera and then taking a picture. If the rod were held in the centre of the picture, perpendicular to the camera lens, then only its end would be visible as a circle on the picture; all other parts of it would be obscured. If you moved it upward, the lower side would start to become visible but you could hide it again by angling the rod upward. If you continued moving the rod up and angling it further upward, the circular end would eventually reach the top edge of the picture. At this point, any object above the line traced by the rod in world space would not be visible on the picture.

The rod could just as easily be moved and rotated left, right, or down or any combination of horizontal and vertical. The angle of the “hidden” rod simply depends on its distance from the centre of the screen in both axes.

The meaning of this thought experiment is that any point in a camera’s image actually corresponds to a line in world space and only a single point along that line is visible in the image. Everything behind that position on the line is obscured.

The outer edges of the image are defined by the diverging lines that correspond to the corners of the image. If those lines were traced backwards towards the camera, they would all eventually converge at a single point. In Unity, this point is located exactly at the camera’s transform position and is known as the centre of perspective. The angle subtended by the lines converging from the top and bottom centres of the screen at the centre of perspective is called the field of view (often abbreviated to FOV).

As stated above, anything that falls outside the diverging lines at the edges of the image will not be visible to the camera, but there are also two other restrictions on what it will render. The near and far **clipping planes** are parallel to the camera’s XY plane and each set at a certain distance along its centre line. Anything closer to the camera than the **near clipping plane** and anything farther away than the **far clipping plane** will not be rendered.



The diverging corner lines of the image along with the two clipping planes define a truncated pyramid - the view frustum.

The Size of the Frustum at a Given Distance from the Camera

Leave feedback

A cross-section of the view frustum at a certain distance from the **camera** defines a rectangle in world space that frames the visible area. It is sometimes useful to calculate the size of this rectangle at a given distance, or find the distance where the rectangle is a given size. For example, if a moving **camera** needs to keep an object (such as the player) completely in shot at all times then it must not get so close that part of that object is cut off.

The height of the frustum at a given distance (both in world units) can be obtained with the following formula:

```
var frustumHeight = 2.0f * distance * Mathf.Tan(camera.fieldOfView * 0.5f * Mat
```

...and the process can be reversed to calculate the distance required to give a specified frustum height:

```
var distance = frustumHeight * 0.5f / Mathf.Tan(camera.fieldOfView * 0.5f * Mat
```

It is also possible to calculate the FOV angle when the height and distance are known:

```
var camera.fieldOfView = 2.0f * Mathf.Atan(frustumHeight * 0.5f / distance) * M
```

Each of these calculations involves the height of the frustum but this can be obtained from the width (and vice versa) very easily:

```
var frustumWidth = frustumHeight * camera.aspect;
var frustumHeight = frustumWidth / camera.aspect;
```

Dolly Zoom (AKA the “Trombone” Effect)

Leave feedback

Dolly Zoom is the well-known visual effect where the **camera** simultaneously moves towards a target object and zooms out from it. The result is that the object appears roughly the same size but all the other objects in the **scene** change perspective. Done subtly, dolly zoom has the effect of highlighting the target object, since it is the only thing in the **scene** that isn’t shifting position in the image. Alternatively, the zoom can be deliberately performed quickly to create the impression of disorientation.

An object that just fits within the frustum vertically will occupy the whole height of the view as seen on the screen. This is true whatever the object’s distance from the camera and whatever the field of view. For example, you can move the camera closer to the object but then widen the field of view so that the object still just fits inside the frustum’s height. That particular object will appear the same size onscreen but everything else will change size as the distance and FOV change. This is the essence of the dolly zoom effect.

Creating the effect in code is a matter of saving the height of the frustum at the object’s position at the start of the zoom. Then, as the camera moves, its new distance is found and the FOV adjusted to keep it the same height at the object’s position. This can be accomplished with the following code:

```
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public Transform target;
    public Camera camera;

    private float initHeightAtDist;
    private bool dzEnabled;

    // Calculate the frustum height at a given distance from the camera.
    void FrustumHeightAtDistance(float distance) {
        return 2.0f * distance * Mathf.Tan(camera.fieldOfView * 0.5f * Mathf.Deg
    }

    // Calculate the FOV needed to get a given frustum height at a given distanc
    void FOVForHeightAndDistance(float height, float distance) {
        return 2.0f * Mathf.Atan(height * 0.5f / distance) * Mathf.Rad2Deg;
    }
```

```
    // Start the dolly zoom effect.
    void StartDZ() {
        var distance = Vector3.Distance(transform.position, target.position);
        initHeightAtDist = FrustumHeightAtDistance(distance);
        dzEnabled = true;
    }

    // Turn dolly zoom off.
    void StopDZ() {
        dzEnabled = false;
    }

    void Start() {
        StartDZ();
    }

    void Update () {
        if (dzEnabled) {
            // Measure the new distance and readjust the FOV accordingly.
            var currDistance = Vector3.Distance(transform.position, target.posit
            camera.fieldOfView = FOVForHeightAndDistance(initHeightAtDist, currD
        }

        // Simple control to allow the camera to be moved in and out using the u
        transform.Translate(Input.GetAxis("Vertical") * Vector3.forward * Time.d
    }
}
```

C# script example

```
var target: Transform;

private var initHeightAtDist: float;
private var dzEnabled: boolean;


// Calculate the frustum height at a given distance from the camera.
function FrustumHeightAtDistance(distance: float) {
    return 2.0 * distance * Mathf.Tan(camera.fieldOfView * 0.5 * Mathf.Deg2Rad);
}
```

```
// Calculate the FOV needed to get a given frustum height at a given distance.
function FOVForHeightAndDistance(height: float, distance: float) {
    return 2 * Mathf.Atan(height * 0.5 / distance) * Mathf.Rad2Deg;
}


// Start the dolly zoom effect.
function StartDZ() {
    var distance = Vector3.Distance(transform.position, target.position);
    initHeightAtDist = FrustumHeightAtDistance(distance);
    dzEnabled = true;
}


// Turn dolly zoom off.
function StopDZ() {
    dzEnabled = false;
}


function Start() {
    StartDZ();
}


function Update () {
    if (dzEnabled) {
        // Measure the new distance and readjust the FOV accordingly.
        var currDistance = Vector3.Distance(transform.position, target.position)
        camera.fieldOfView = FOVForHeightAndDistance(initHeightAtDist, currDista
    }

    // Simple control to allow the camera to be moved in and out using the up/do
    transform.Translate(Input.GetAxis("Vertical") * Vector3.forward * Time.delta
}
```

JS script example

Rays from the Camera

Leave feedback

In the section Understanding the View Frustum, it was explained that any point in the **camera**'s view corresponds to a line in world space. It is sometimes useful to have a mathematical representation of that line and Unity can provide this in the form of a Ray object. The Ray always corresponds to a point in the view, so the Camera class provides the ScreenPointToRay and ViewportPointToRay functions. The difference between the two is that ScreenPointToRay expects the point to be provided as a **pixel** coordinate, while ViewportPointToRay takes normalized coordinates in the range 0..1 (where 0 represents the bottom or left and 1 represents the top or right of the view). Each of these functions returns a Ray which consists of a point of origin and a vector which shows the direction of the line from that origin. The Ray originates from the near **clipping plane** rather than the Camera's transform.position point.

Raycasting

The most common use of a Ray from the camera is to perform a raycast out into the **scene**. A raycast sends an imaginary "laser beam" along the ray from its origin until it hits a **collider** in the **scene**. Information is then returned about the object and the point that was hit in a RaycastHit object. This is a very useful way to locate an object based on its onscreen image. For example, the object at the mouse position can be determined with the following code:

C# script example:

```
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public Camera camera;

    void Start(){
        RaycastHit hit;
        Ray ray = camera.ScreenPointToRay(Input.mousePosition);

        if (Physics.Raycast(ray, out hit)) {
            Transform objectHit = hit.transform;

            // Do something with the object that was hit by the raycast.
        }
    }
}
```

JS script example:

```
var hit: RaycastHit;
var ray: Ray = camera.ScreenPointToRay(Input.mousePosition);
```

```
if (Physics.Raycast(ray, hit)) {
    var objectHit: Transform = hit.transform;

    // Do something with the object that was hit by the raycast.
}
```

Moving the Camera Along a Ray

It is sometimes useful to get a ray corresponding to a screen position and then move the camera along that ray. For example, you may want to allow the user to select an object with the mouse and then zoom in on it while keeping it "pinned" to the same screen position under the mouse (this might be useful when the camera is looking at a tactical map, for example). The code to do this is fairly straightforward:

C# script example:

```
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public bool zooming;
    public float zoomSpeed;
    public Camera camera;

    void Update() {
        if (zooming) {
            Ray ray = camera.ScreenPointToRay(Input.mousePosition);
            float zoomDistance = zoomSpeed * Input.GetAxis("Vertical") * Time.delt
            camera.transform.Translate(ray.direction * zoomDistance, Space.World)
        }
    }
}
```

JS script example:

```
var zooming: boolean;
var zoomSpeed: float;

if (zooming) {
    var ray: Ray = camera.ScreenPointToRay(Input.mousePosition);
    zoomDistance = zoomSpeed * Input.GetAxis("Vertical") * Time.deltaTime;
```

```
    camera.transform.Translate(ray.direction * zoomDistance, Space.World);
}
```

Using an Oblique Frustum

Leave feedback

By default, the view frustum is arranged symmetrically around the **camera**'s centre line but it doesn't necessarily need to be. The frustum can be made "oblique", which means that one side is at a smaller angle to the centre line than the opposite side. The effect is rather like taking a printed photograph and cutting one edge off. This makes the perspective on one side of the image seem more condensed giving the impression that the viewer is very close to the object visible at that edge. An example of how this can be used is a car racing game where the frustum might be flattened at its bottom edge. This would make the viewer seem closer to the road, accentuating the feeling of speed.

While the camera class doesn't have functions to set the obliqueness of the frustum, it can be done quite easily by altering the projection matrix:

```
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void SetObliqueness(float horizObl, float vertObl) {
        Matrix4x4 mat  = Camera.main.projectionMatrix;
        mat[0, 2] = horizObl;
        mat[1, 2] = vertObl;
        Camera.main.projectionMatrix = mat;
    }
}
```

C# script example

```
function SetObliqueness(horizObl: float, vertObl: float) {
    var mat: Matrix4x4 = camera.projectionMatrix;
    mat[0, 2] = horizObl;
    mat[1, 2] = vertObl;
    camera.projectionMatrix = mat;
}
```

JS script example

Mercifully, it is not necessary to understand how the projection matrix works to make use of this. The horizObl and vertObl values set the amount of horizontal and vertical obliqueness, respectively. A value of zero indicates no obliqueness. A positive value shifts the frustum rightwards or upwards, thereby flattening the left or bottom side. A negative value shifts leftwards or downwards and consequently flattens the right or top side of the frustum. The effect can be seen directly if this script is added to a camera and the game is switched to the **scene view** while the game runs; the wireframe depiction of the camera's frustum will change as you vary the values of horizObl and vertObl in the **inspector**. A value of 1 or –1 in either variable indicates that one side of the frustum is completely flat against the centreline. It is possible although seldom necessary to use values outside this range.

Creating an Impression of Large or Small Size

Leave feedback

From the graphical point of view, the units of distance in Unity are arbitrary and don't correspond to real world measurements. Although this gives flexibility and convenience for design, it is not always easy to convey the intended size of the object. For example, a toy car looks different to a full size car even though it may be an accurate scale model of the real thing.

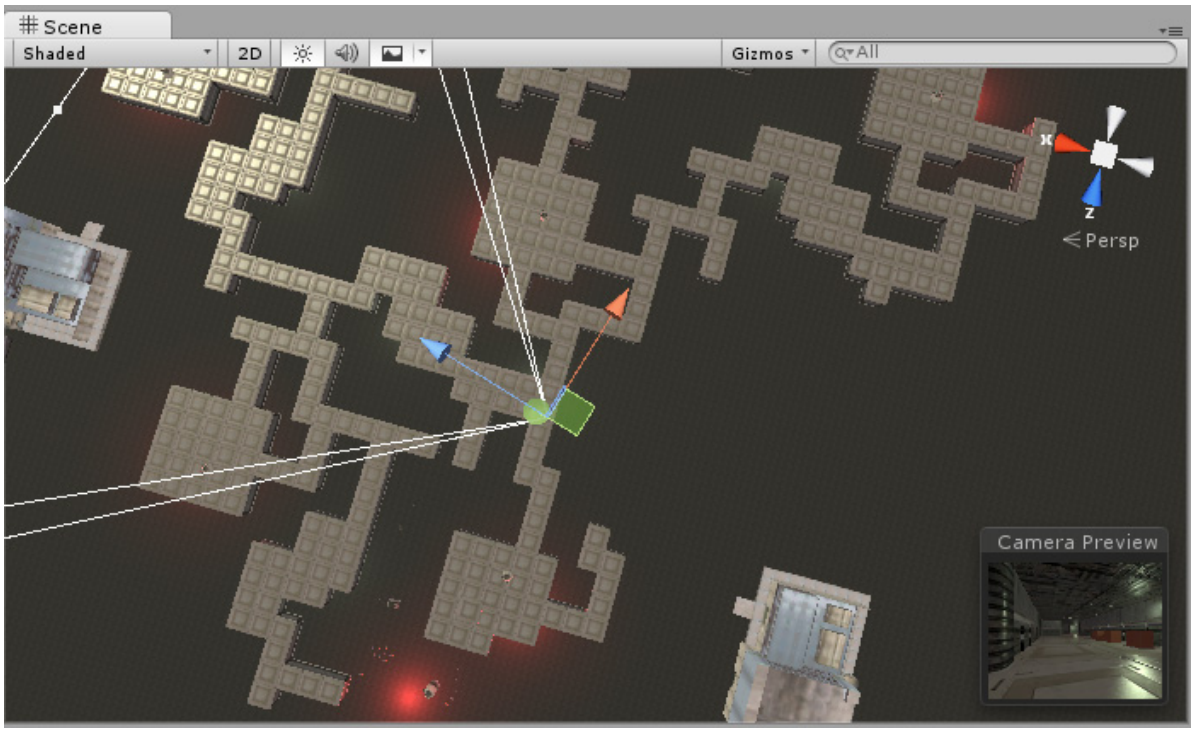
A major element in the impression of an object's size is the way the perspective changes over the object's length. For example, if a toy car is viewed from behind then the front of the car will only be a short distance farther away than the back. Since the distance is small, perspective will have relatively little effect and so the front will appear little different in size to the back. With a full size car, however, the front will be several metres farther away from the **camera** than the back and the effect of perspective will be much more noticeable.

For an object to appear small, the lines of perspective should diverge only very slightly over its depth. You can achieve this by using a narrower field of view than the default 60 degrees and moving the camera farther away to compensate for the increased onscreen size. Conversely, if you want to make an object look big, use a wide FOV and move the camera in close. When these perspective alterations are used with other obvious techniques (like looking down at a "small" object from higher-than-normal vantage point) the result can be quite convincing.

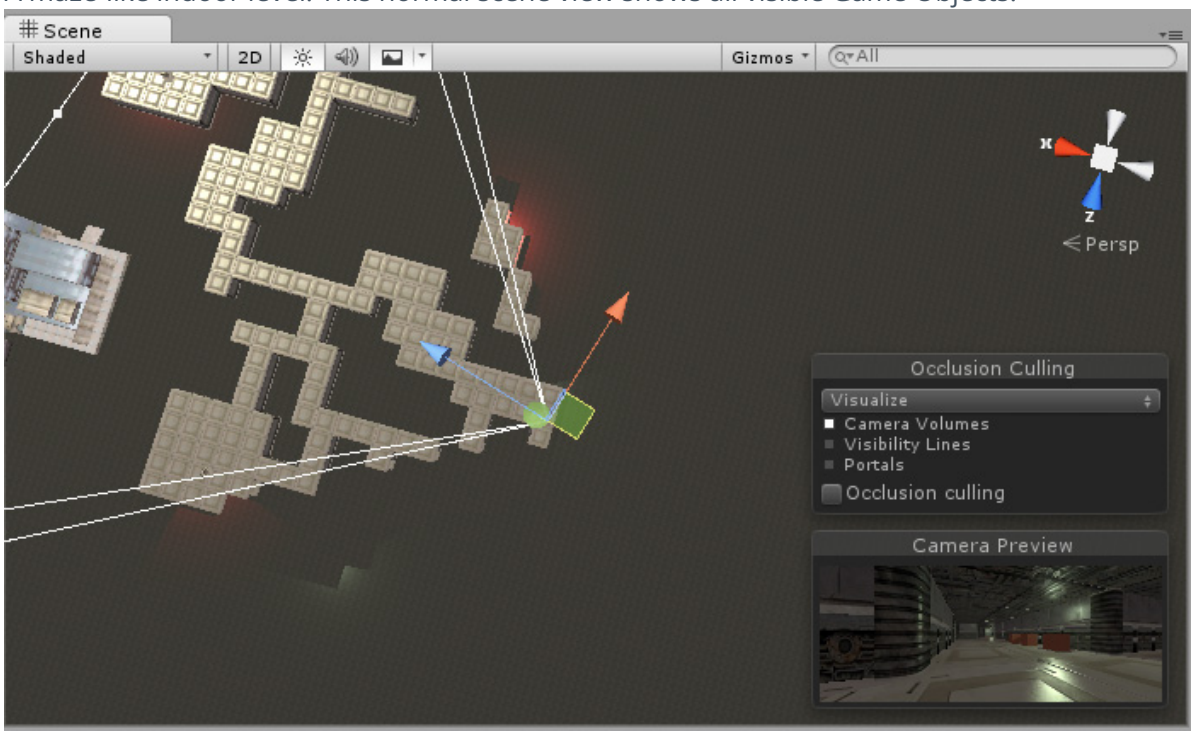
Occlusion Culling

Leave feedback

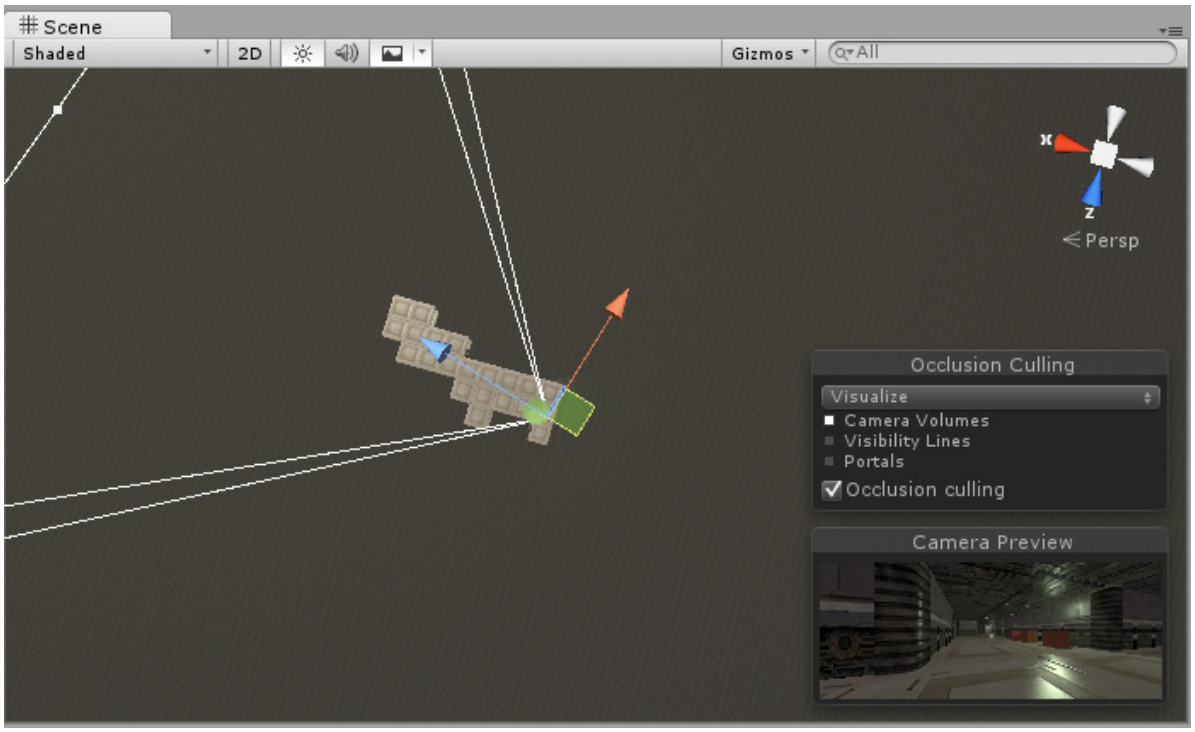
Occlusion Culling is a feature that disables rendering of objects when they are not currently seen by the **camera** because they are obscured (occluded) by other objects. This does not happen automatically in 3D computer graphics since most of the time objects farthest away from the camera are drawn first and closer objects are drawn over the top of them (this is called “overdraw”). Occlusion Culling is different from Frustum Culling. Frustum Culling only disables the renderers for objects that are outside the camera’s viewing area but does not disable anything hidden from view by overdraw. Note that when you use Occlusion Culling you will still benefit from Frustum Culling.



A maze-like indoor level. This normal scene view shows all visible Game Objects.



Regular frustum culling only renders objects within the camera’s view. This is automatic and always happens.



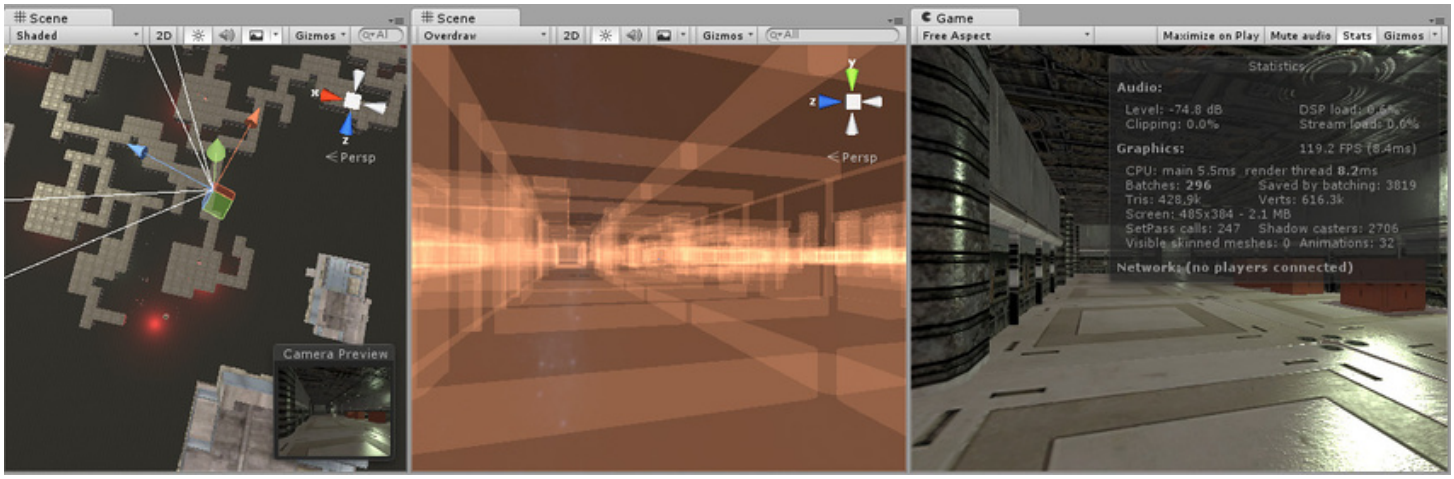
Occlusion culling removes additional objects from within the camera rendering work if they are entirely obscured by nearer objects.

The occlusion culling process will go through the **scene** using a virtual camera to build a hierarchy of potentially visible sets of objects. This data is used at runtime by each camera to identify what is visible and what is not. Equipped with this information, Unity will ensure only visible objects get sent to be rendered. This reduces the number of draw calls and increases the performance of the game.

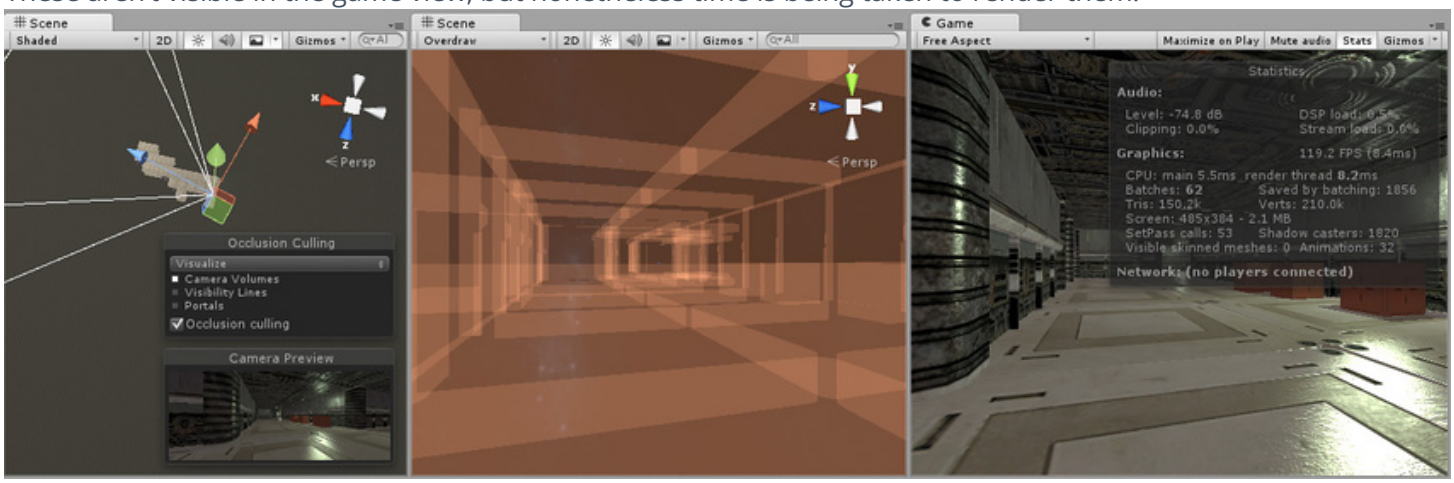
The data for occlusion culling is composed of cells. Each cell is a subdivision of the entire **bounding volume** of the scene. More specifically the cells form a binary tree. Occlusion Culling uses two trees, one for View Cells (Static Objects) and the other for Target Cells (Moving Objects). View Cells map to a list of indices that define the visible static objects which gives more accurate culling results for static objects.

It is important to keep this in mind when creating your objects because you need a good balance between the size of your objects and the size of the cells. Ideally, you shouldn't have cells that are too small in comparison with your objects but equally you shouldn't have objects that cover many cells. You can sometimes improve the culling by breaking large objects into smaller pieces. However, you can still merge small objects together to reduce draw calls and, as long as they all belong to the same cell, occlusion culling will not be affected.

You can use the 'overdraw' scene **rendering mode** to see the amount of overdraw that is occuring, and the stats information pane in the game view to see the amount of triangles, verts, and batches that are being rendered. Below is a comparison of these before and after applying occlusion culling.



Notice in the Overdraw scene view, a high density of overdraw as many rooms beyond the visible walls are rendered. These aren't visible in the game view, but nonetheless time is being taken to render them.

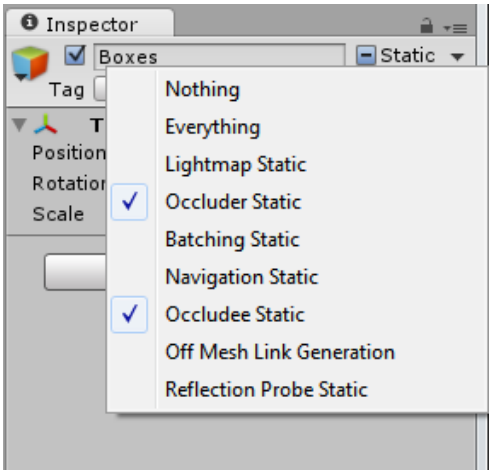


With occlusion culling applied, the distant rooms are not rendered, the overdraw is much less dense, and the number of triangles and batches being rendered has dropped dramatically, without any change to how the game view looks.

Setting up Occlusion Culling

In order to use Occlusion Culling, there is some manual setup involved. First, your level geometry must be broken into sensibly sized pieces. It is also helpful to lay out your levels into small, well defined areas that are occluded from each other by large objects such as walls, buildings, **etc**. The idea here is that each individual **mesh** will be turned on or off based on the occlusion data. So if you have one object that contains all the furniture in your room then either all or none of the entire set of furniture will be culled. This doesn't make nearly as much sense as making each piece of furniture its own **mesh**, so each can individually be culled based on the camera's view point.

You need to tag all scene objects that you want to be part of the occlusion to **Occluder Static** in the **Inspector**. The fastest way to do this is to multi-select the objects you want to be included in occlusion calculations, and mark them as **Occluder Static** and **Occludee Static**.



Marking an object for Occlusion

When should you use **Occludee Static**? Completely transparent or translucent objects that do not occlude, as well as small objects that are unlikely to occlude other things, should be marked as **Occludees**, but not **Occluders**. This means they will be considered in occlusion by other objects, but will not be considered as occluders themselves, which will help reduce computation.

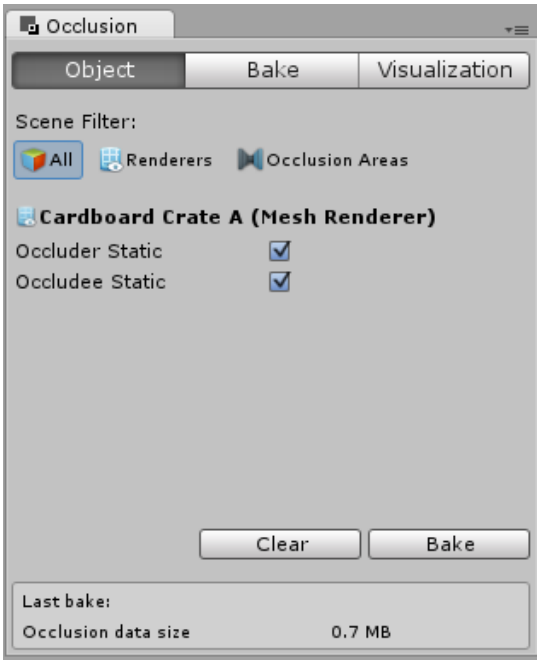
When using **LOD groups**, only the base level object (LOD0) may be used as an Occluder.

Occlusion Culling Window

For most operations dealing with Occlusion Culling, you should use the Occlusion Culling Window (**Window** > **Rendering** > **Occlusion Culling**)

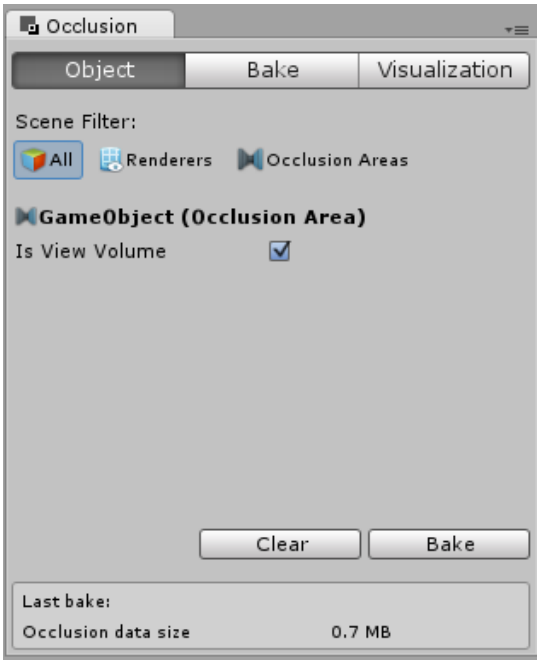
In the Occlusion Culling Window, you can work with occluder meshes, and Occlusion Areas.

If you are in the **Object** tab of the **Occlusion Culling Window** and have a Mesh Renderer selected in the scene, you can modify the relevant Static flags:



Occlusion Culling Window for a Mesh Renderer

If you are in the **Object** tab of the **Occlusion Culling Window** and have an Occlusion Area selected, you can work with relevant OcclusionArea properties (for more details go to the Occlusion Area section)

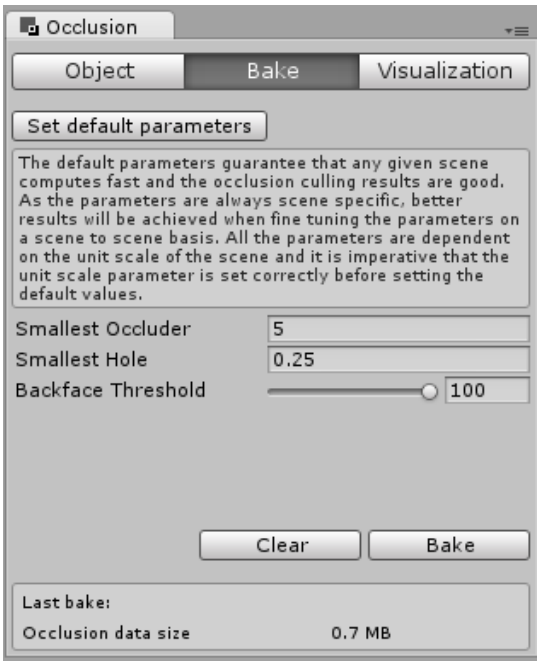


Occlusion Culling Window for the Occlusion Area

NOTE: By default if you don't create any occlusion areas, occlusion culling will be applied to the whole scene.

NOTE: Whenever your camera is outside occlusion areas, occlusion culling will not be applied. It is important to set up your Occlusion Areas to cover the places where the camera can potentially be, but making the areas too large incurs a cost during baking.

Occlusion Culling - Bake



Occlusion culling inspector bake tab.

The occlusion culling bake window has a "Set Default Parameters" button, which allows you to reset the bake values to Unity's default values. These are good for many typical scenes, however you'll often be able to get better results by adjusting the values to suit the particular contents of your scene.

Properties

Property	Function
Smallest Occluder	The size of the smallest object that will be used to hide other objects when doing occlusion culling. Any objects smaller than this size will never cause objects occluded by them to be culled. For example, with a value of 5, all objects that are higher or wider than 5 meters will cause hidden objects behind them to be culled (not rendered, saving render time). Picking a good value for this property is a balance between occlusion accuracy and storage size for the occlusion data.
Smallest Hole	This value represents the smallest gap between geometry through which the camera is supposed to see. The value represents the diameter of an object that could fit through the hole. If your scene has very small cracks through which the camera should be able to see, the Smallest Hole value must be smaller than the narrowest dimension of the gap.
Backface Threshold	Unity's occlusion uses a data size optimization which reduces unnecessary details by testing backfaces. The default value of 100 is robust and never removes backfaces from the dataset. A value of 5 would aggressively reduce the data based on locations with visible backfaces. The idea is that typically, valid camera positions would not normally see many backfaces - for example, the view of the underside of a **terrain**, or the view from within a solid object that you should not be able to reach. With a threshold lower than 100, Unity will remove these areas from the dataset entirely, thereby reducing the data size for the occlusion.

At the bottom of the bake tab is are the Clear and Bake buttons. Click on the **Bake** Button to start generating the Occlusion Culling data. Once the data is generated, you can use the Visualization tab to preview and test the occlusion culling. If you are not satisfied with the results, click on the **Clear** button to remove previously calculated data, adjust the settings, and bake again.

Occlusion Culling - Visualization

Occlusion culling inspector visualization tab.

All the objects in the scene affect the size of the bounding volume so try to keep them all within the visible bounds of the scene.

When you're ready to generate the occlusion data, click the **Bake** button. Remember to choose the **Memory Limit** in the **Bake** tab. Lower values make the generation quicker and less precise, higher values are to be used for production quality closer to release.

Bear in mind that the time taken to build the occlusion data will depend on the cell levels, the data size and the quality you have chosen.

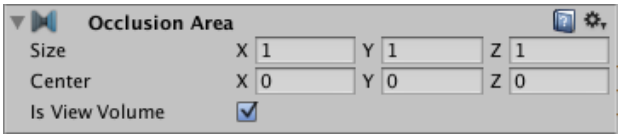
After the processing is done, you should see some colorful cubes in the View Area. The colored areas are regions that share the same occlusion data.

Click on **Clear** if you want to remove all the pre-calculated data for Occlusion Culling.

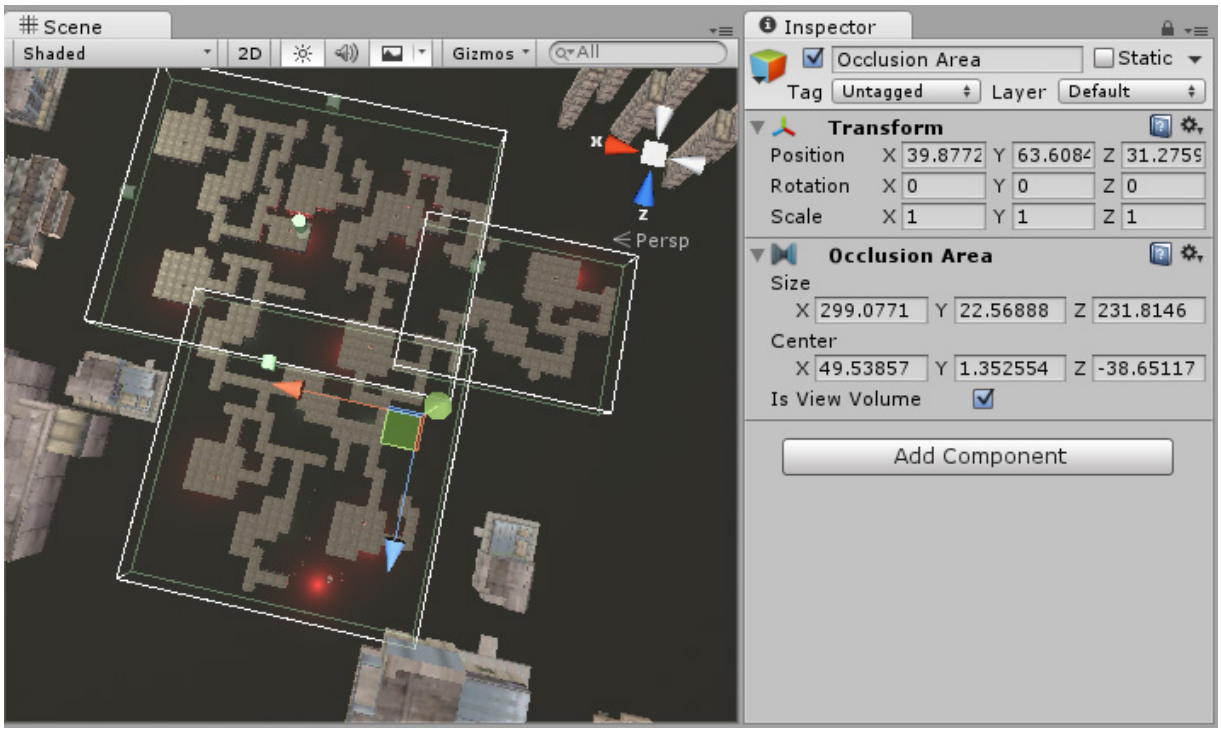
Occlusion Area

To apply occlusion culling to moving objects you have to create an **Occlusion Area** and then modify its size to fit the space where the moving objects will be located (of course the moving objects cannot be marked as static). You can create Occlusion Areas by adding the **Occlusion Area** component to an empty game object (**Component -> Rendering -> Occlusion Area** in the menus).

After creating the **Occlusion Area**, check the *Is View Volume* checkbox to occlude moving objects.

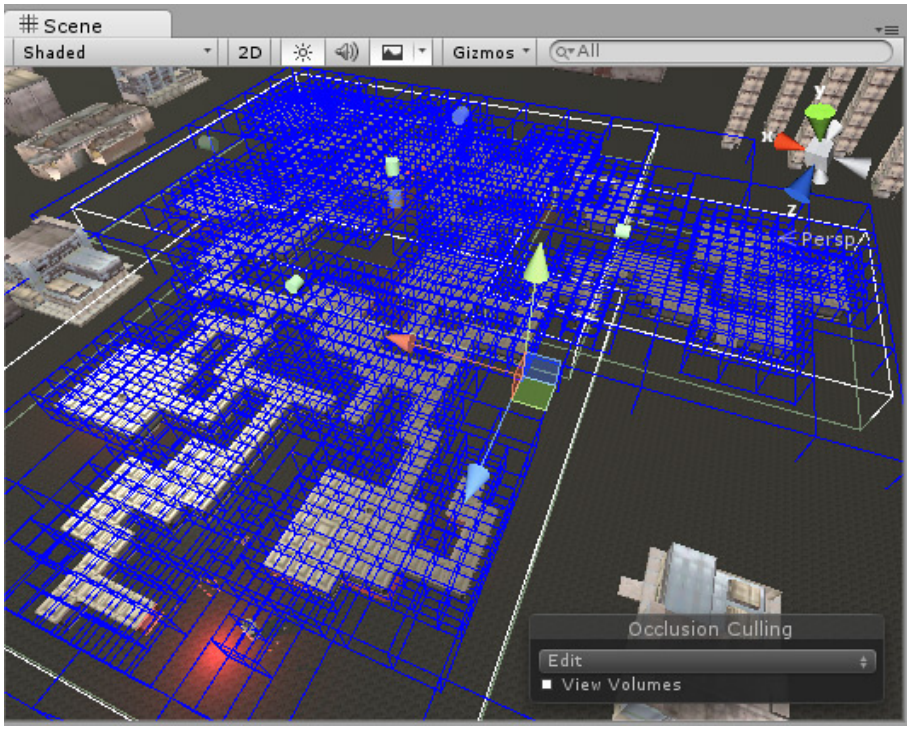


Property:	*Function:*
Size	Defines the size of the Occlusion Area.
Center	Sets the center of the Occlusion Area. By default this is 0,0,0 and is located in the center of the box.
Is View Volume	Defines where the camera can be. Check this in order to occlude static objects that are inside this *Occlusion Area*.



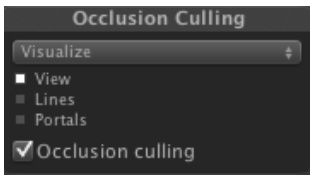
Occlusion Area properties for moving objects.

After you have added the Occlusion Area, you need to see how it divides the box into cells. To see how the occlusion area will be calculated, select **Edit** and toggle the **View** button in the **Occlusion Culling Preview Panel**.



Testing the generated occlusion

After your occlusion is set up, you can test it by enabling the *Occlusion Culling* (in the **Occlusion Culling Preview Panel** in Visualize mode) and moving the **Main Camera** around in the **scene view**.



The Occlusion View mode in Scene View

As you move the Main Camera around (whether or not you are in Play mode), you'll see various objects disable themselves. The thing you are looking for here is any error in the occlusion data. You'll recognize an error if you see objects suddenly popping into view as you move around. If this happens, your options for fixing the error are either to change the resolution (if you are playing with target volumes) or to move objects around to cover up the error. To debug problems with occlusion, you can move the Main Camera to the problematic position for spot-checking.

When the processing is done, you should see some colorful cubes in the View Area. The blue cubes represent the cell divisions for **Target Volumes**. The white cubes represent cell divisions for **View Volumes**. If the parameters were set correctly you should see some objects not being rendered. This will be because they are either outside of the view frustum of the camera or else occluded from view by other objects.

After occlusion is completed, if you don't see anything being occluded in your scene then try breaking your objects into smaller pieces so they can be completely contained inside the cells.

Materials, Shaders & Textures

Leave feedback

Rendering in Unity uses **Materials**, **Shaders** and **Textures**. All three have a close relationship.

Materials define how a surface should be rendered, by including references to the Textures it uses, tiling information, Color tints and more. The available options for a Material depend on which Shader the Material is using.

Shaders are small **scripts** that contain the mathematical calculations and algorithms for calculating the Color of each **pixel** rendered, based on the lighting input and the Material configuration.

Textures are bitmap images. A Material can contain references to textures, so that the Material's Shader can use the textures while calculating the surface color of a **GameObject**. In addition to basic Color (Albedo) of a GameObject's surface, Textures can represent many other aspects of a Material's surface such as its reflectivity or roughness.

A Material specifies one specific Shader to use, and the Shader used determines which options are available in the Material. A Shader specifies one or more Texture variables that it expects to use, and the Material **Inspector** in Unity allows you to assign your own Texture **Assets** to these Texture variables.

For most normal **rendering** (such as rendering characters, scenery, environments, solid and transparent GameObjects, hard and soft surfaces) the Standard Shader is usually the best choice. This is a highly customisable shader which is capable of rendering many types of surface in a highly realistic way.

There are other situations where a different built-in Shader, or even a custom written shader might be appropriate (for example liquids, foliage, refractive glass, particle effects, cartoony, illustrative or other artistic effects, or other special effects like night vision, heat vision or x-ray vision).

For more information, see the following pages:

Creating and Using Materials
The Built-in Standard Shader
Other built-in Shaders
Writing Your Own Shaders

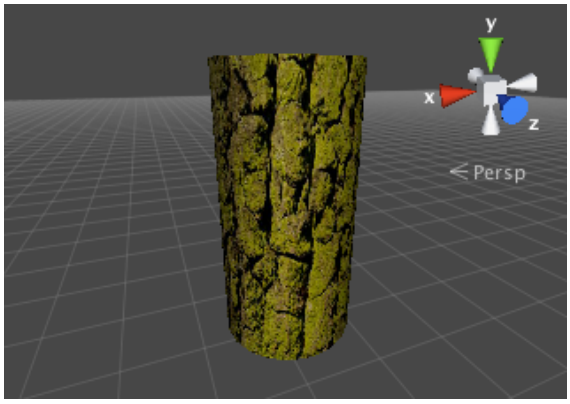
2017–10–26 New page

2017–10–26 Page amended with limited editorial review

Textures

Leave feedback

Normally, the **mesh** geometry of an object only gives a rough approximation of the shape while most of the fine detail is supplied by **Textures**. A texture is just a standard bitmap image that is applied over the **mesh** surface. You can think of a texture image as though it were printed on a rubber sheet that is stretched and pinned onto the **mesh** at appropriate positions. The positioning of the texture is done with the 3D modelling software that is used to create the **mesh**.



Cylinder with tree bark

Unity can import textures from most common image file formats.

Textures for use on 3D models

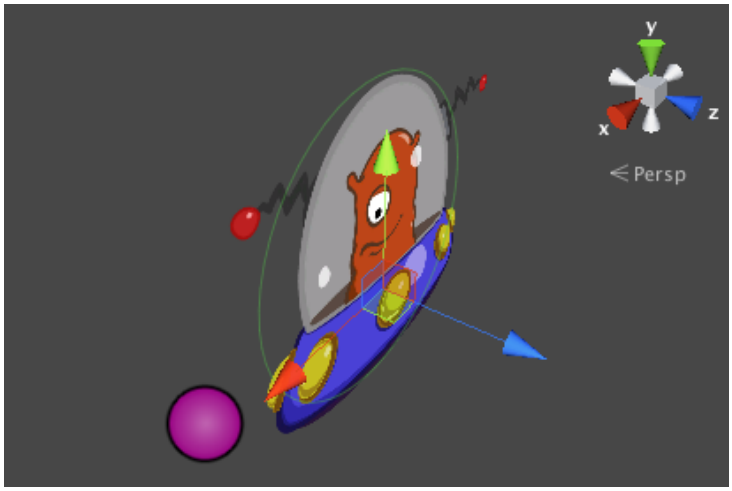
Textures are applied to objects using Materials. Materials use specialised graphics programs called Shaders to render a texture on the mesh surface. Shaders can implement lighting and colouring effects to simulate shiny or bumpy surfaces among many other things. They can also make use of two or more textures at a time, combining them for even greater flexibility.

You should make your textures in dimensions that are to the power of two (e.g. 32x32, 64x64, 128x128, 256x256, etc.) Simply placing them in your project's **Assets** folder is sufficient, and they will appear in the **Project View**.

Once your texture has been imported, you should assign it to a Material. The material can then be applied to a mesh, **Particle System**, or **GUI Texture**. Using the **Import Settings**, it can also be converted to a **Cubemap** or **Normalmap** for different types of applications in the game. For more information about importing textures, please read the Texture Component page.

2D graphics

In 2D games, the **Sprites** are implemented using textures applied to flat meshes that approximate the objects' shapes.



Sprite from a 3D viewpoint

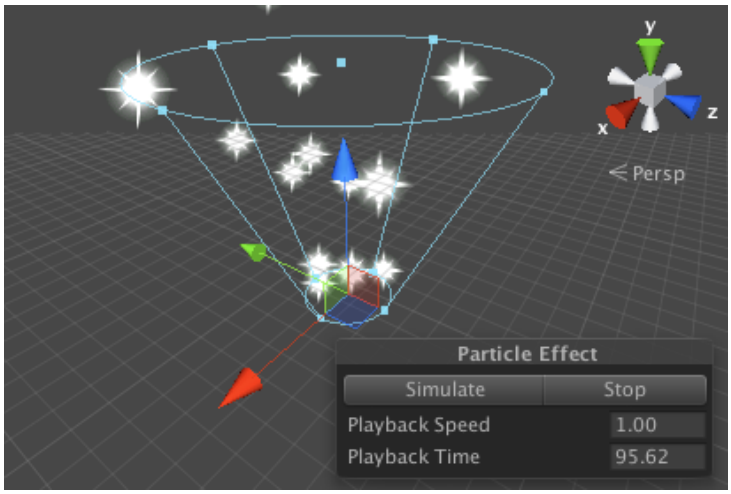
An object in a 2D game may require a set of related graphic images to represent animation frames or different states of a character. Special techniques are available to allow these sets of images to be designed and rendered efficiently. See the manual page about the Sprite Editor for more information.

GUI

A game's *graphic user interface* (GUI) consists of graphics that are not used directly in the game **scene** but are there to allow the player to make choices and see information. For example, the score display and the options menu are typical examples of game GUI. These graphics are clearly very different from the kind used to detail a mesh surface but they are handled using standard Unity textures nevertheless. See the manual chapter on GUI Scripting Guide for further details about Unity's GUI system.

Particles

Meshes are ideal for representing solid objects but less suited for things like flames, smoke and sparkles left by a magic spell. This type of effect is handled much better by **Particle Systems**. A *particle* is a small 2D graphic representing a small portion of something that is basically fluid or gaseous, such as a smoke cloud. When many of these particles are created at once and set in motion, optionally with random variations, they can create a very convincing effect. For example, you might display an explosion by sending particles with a fire texture out at great speed from a central point. A waterfall could be simulated by accelerating water particles downward from a line high in the scene.



Star particle system

Unity's particle systems have a wealth of options for creating all kinds of fluid effects. See the manual chapter on the subject for further information.

Terrain Heightmaps

Textures can even be used in cases where the image will never be viewed at all, at least not directly. In a greyscale image, each **pixel** value is simply a number corresponding to the shade of grey at that point in the image (this could be a value in the range 0..1 where zero is black and one is white, say). Although an image like this can be viewed, there is no reason why the numeric pixel values can't be used for other purposes as well, and this is precisely what is done with **Terrain Heightmaps**.

A *terrain* is a mesh representing an area of ground where each point on the ground has a particular height from a baseline. The *heightmap* for a **terrain** stores the numeric height samples at regular intervals as greyscale values in an image where each pixel corresponds to a grid coordinate on the ground. The values are not shown in the scene as an image but are converted to coordinates that are used to generate the **terrain** mesh.

Interestingly, even though a heightmap is not viewed directly as an image, there are still common image processing techniques that are useful when applied to the height data. For example, adding noise to a heightmap will create the impression of rocky terrain while blurring will smooth it out to produce a softer, rolling landscape.

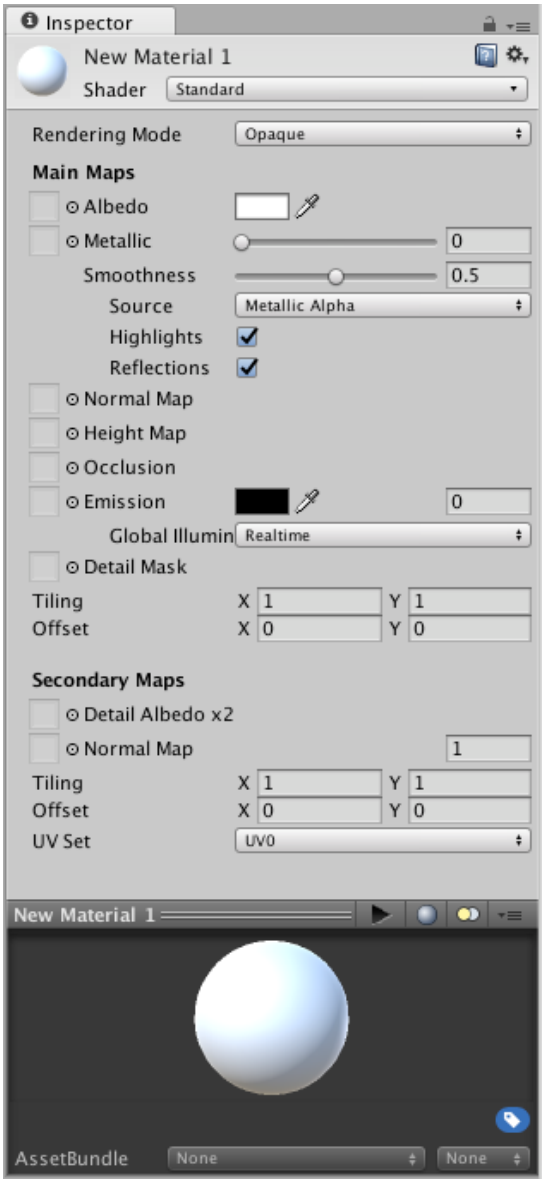
More information about terrains in Unity can be found in this section of the manual.

Creating and Using Materials

Leave feedback

To create a new Material, use **Assets->Create->Material** from the main menu or the **Project View** context menu.

By default, new materials are assigned the Standard Shader, with all map properties empty, like this:



Once the Material has been created, you can apply it to an object and tweak all of its properties in the **Inspector**. To apply it to an object, just drag it from the **Project View** to any object in the **Scene** or **Hierarchy**.

Setting Material Properties

You can select which Shader you want any particular Material to use. Simply expand the **Shader** drop-down in the Inspector, and choose your new Shader. The Shader you choose will dictate the available properties to change. The properties can be colors, sliders, textures, numbers, or vectors. If you have applied the Material to an active object in the **Scene**, you will see your property changes applied to the object in real-time.

There are two ways to apply a **Texture** to a property.

Drag it from the Project View on top of the Texture square
Click the **Select** button, and choose the texture from the drop-down list that appears

Built-in Shaders

In addition to the Standard Shader, there are a number of other categories of built-in shaders for specialised purposes:

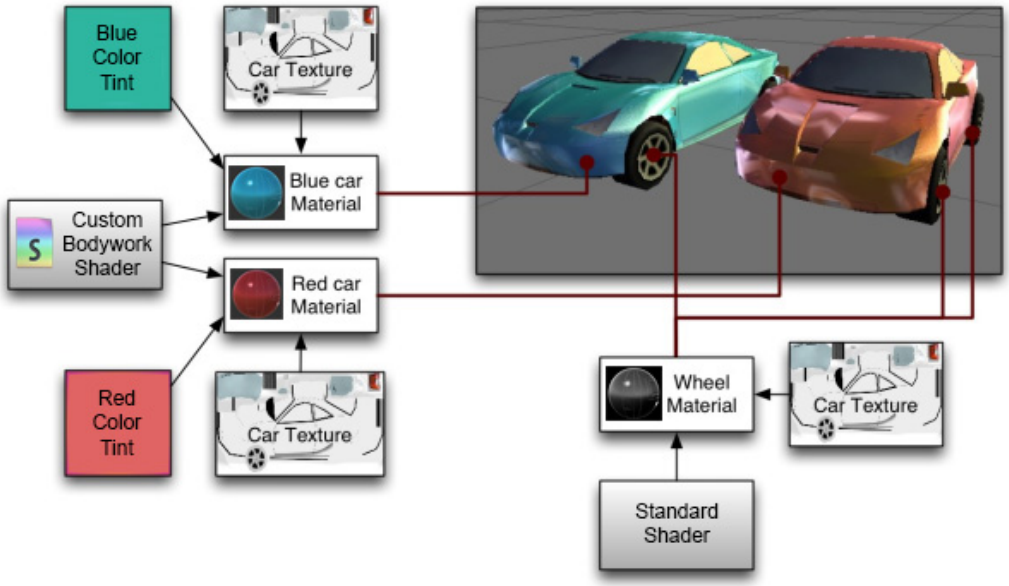
FX: Lighting and glass effects.
GUI and **UI** : For user interface graphics.
Mobile: Simplified high-performance shader for mobile devices.
Nature: For trees and **terrain**.
Particles: **Particle system** effects.
Skybox : For **rendering** background environments behind all geometry
Sprites : For use with the 2D sprite system
Toon: Cartoon-style rendering.
Unlit: For rendering that entirely bypasses all light & shadowing
Legacy: The large collection of older shaders which were superseded by the Standard Shader

Shader technical details

A Shader is a script which contains mathematical calculations and algorithms for how the **pixels** on the surface of a model should look. The standard shader performs complex and realistic lighting calculations. Other shaders may use simpler or different calculations to show different results. Within any given Shader are a number of properties which can be given values by a Material using that shader. These properties can be numbers, colours definitions or textures, which appear in the inspector when viewing a Material. Materials are then used by Renderer **components** attached to Game Objects, to render each Game Object's **mesh**.

It is possible and often desirable to have several different Materials which may reference the same textures. These materials may also use the same or different shaders, depending on the requirements.

Below is an example of a possible set-up combination using three materials, two shaders and one texture.



In the diagram we have a red car and a blue car. Both models use a separate material for the bodywork, "Red car material" and "Blue car material" respectively.

Both these bodywork materials use the same custom shader, "Carbody Shader". A custom shader may be used because the shader adds extra features specifically for the cars, such as metallic sparkly rendering, or perhaps has a custom damage masking feature.

Each car body material has a reference to the "Car Texture", which is a texture map containing all the details of the bodywork, without a specific paint colour.

The Carbody shader also accepts a tint colour, which is set to a different colour for the red and blue cars, giving each car a different look while using a single texture for both of them.

The car wheel models use a separate material again, but this time both cars share the same material for their wheels, as the wheels do not differ on each car. The wheel material uses the Standard Shader, and has a reference again to the Car Texture.

Notice how the car texture contains details for the bodywork *and* wheels - this is a *texture atlas*, meaning different parts of the texture image are explicitly mapped to different parts of the model.

Even though the bodywork materials are using a texture that also contains the wheel image, the wheel does not appear on the body because that part of the texture is not mapped to the bodywork geometry.

Similarly, the wheel material is using the same texture, which has bodywork detail in it. The bodywork detail does not appear on the wheel, because only the portion of the texture showing the wheel detail is mapped to the wheel geometry.

This mapping is done by the 3D artist in an external 3d application, and is called “UV mapping”.

To be more specific, a Shader defines:

The method to render an object. This includes code and mathematical calculations that may include the angles of light sources, the viewing angle, and any other relevant calculations. Shaders can also specify different methods depending on the graphics hardware of the end user.
The parameters that can be customised in the material inspector, such as texture maps, colours and numeric values.
A Material defines:

Which shader to use for rendering this material.
The specific values for the shader’s parameters - such as which texture maps, the colour and numeric values to use.
Custom Shaders are meant to be written by graphics programmers. They are created using the **ShaderLab** language, which is quite simple. However, getting a shader to work well on a variety graphics cards is an involved job and requires a fairly comprehensive knowledge of how graphics cards work.

A number of shaders are built into Unity directly, and some more come in the Standard Assets Library.

Standard Shader

Leave feedback

The Unity **Standard Shader** is a built-in **shader** with a comprehensive set of features. It can be used to render "real-world" objects such as stone, wood, glass, plastic and metal, and supports a wide range of **shader** types and combinations. Its features are enabled or disabled by simply using or not using the various texture slots and parameters in the material editor.

The Standard Shader also incorporates an advanced lighting model called **Physically Based Shading**. Physically Based Shading (PBS) simulates the interactions between materials and light in a way that mimics reality. PBS has only recently become possible in real-time graphics. It works at its best in situations where lighting and materials need to exist together intuitively and realistically.

The idea behind our Physically Based Shader is to create a user-friendly way of achieving a consistent, plausible look under different lighting conditions. It models how light behaves in reality, without using multiple ad-hoc models that may or may not work. To do so, it follows principles of physics, including energy conservation (meaning that objects never reflect more light than they receive), Fresnel reflections (all surfaces become more reflective at grazing angles), and how surfaces occlude themselves (what is called Geometry Term), among others.

The Standard Shader is designed with hard surfaces in mind (also known as "architectural materials"), and can deal with most real-world materials like stone, glass, ceramics, brass, silver or rubber. It will even do a decent job with non-hard materials like skin, hair and cloth.

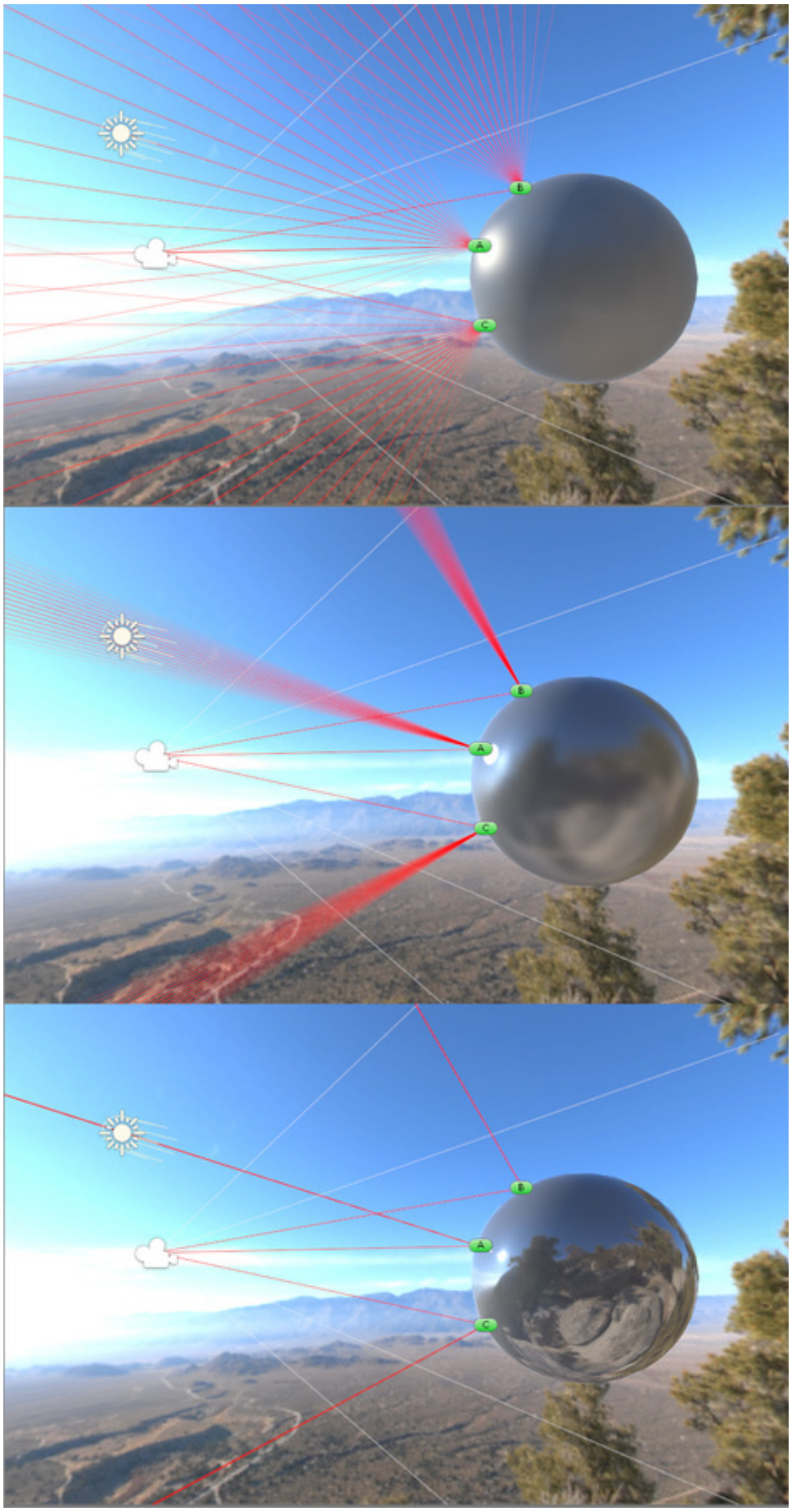
A scene rendered using the standard shader on all models

With the Standard Shader, a large range of shader types (such as Diffuse, Specular, Bumped Specular, Reflective) are combined into a single shader intended to be used across all material types. The benefit of this is that the same lighting calculations are used in all areas of your **scene**, which gives a realistic, consistent and believable distribution of light and shade across all models that use the shader.

Terminology

There are a number of concepts that are very useful when talking about Physically Based Shading in Unity. These include:

Energy conservation - This is a physics-based concept that ensures objects never reflect more light than they receive. The more specular a material is, the less diffuse it should be; the smoother a surface is, the stronger and smaller the highlight gets.



The light rendered at each point on a surface is calculated to be the same as the amout of light received from its environment. The microfacets of rough surfaces are affected by light from a wider area. Smoother surfaces give stronger and smaller highlights. Point A reflects light from the source

towards the camera. Point B takes on a blue tint from ambient light from the sky. Point C takes its ambient and reflective lighting from the surrounding ground colours.

High Dynamic Range (HDR) - This refers to colours outside the usual 0–1 range. For instance, the sun can easily be ten times brighter than a blue sky. For an in-depth discussion, see the Unity Manual HDR page.

A scene using High Dynamic Range. The sunlight reflecting in the car window appears far brighter than other objects in the scene, because it has been processed using HDR

Content and Context

Leave feedback

When thinking about lighting in Unity, it is handy to divide the concepts into what we call the **content** the item being lit and rendered, and the **context**, which is the lighting that exists in the **scene** which affects the object being lit.

The Context

When lighting an object it is important to understand which sources of light are affecting the object. There are usually direct light sources in your scene: Game Object lights that you may have placed around your scene. There are also indirect light sources such as reflections and bounced light. These all have an effect on the object's material to give the final result that the **camera** sees across the surface of the object.

This is not a hard and fast separation, often what might be considered "content" could also be the part of the lighting context for another object.

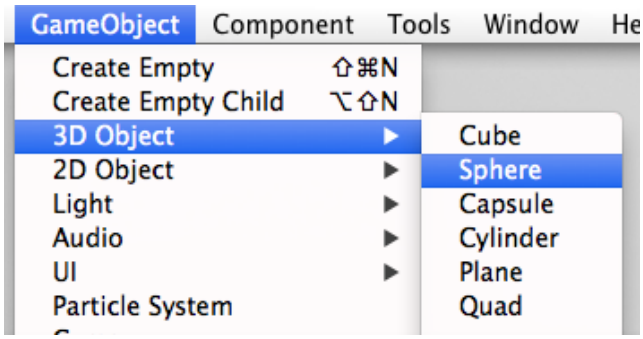
A good example of this would be a building situated in a desert landscape. The building would take light information from the **skybox**, and perhaps bounced light from the surrounding ground.

However there may be a character standing near an exterior wall of the building. For this character, the building is part of the lighting context - it may be casting a shadow, it may be casting bounced light from its wall onto the character, or the character may have reflective parts which are directly reflecting the building itself.

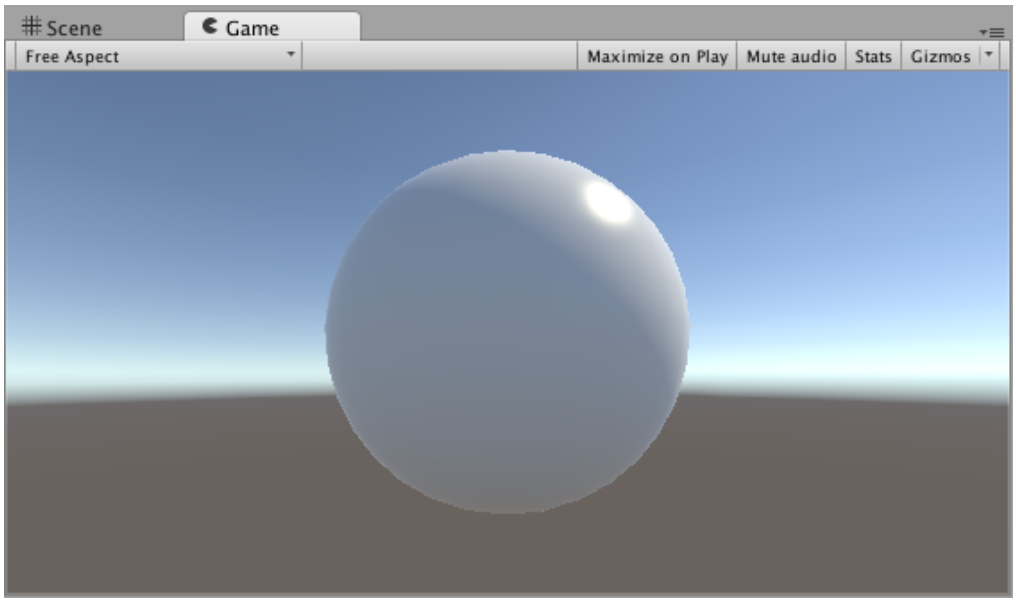
The Default Lighting Context

At startup, Unity 5 shows an empty scene. This scene already has a default lighting context available with ambient, skydome-based reflections and a directional light. Any object placed in that scene should, by default, have all lighting it needs to look correct.

Let's add a sphere to the scene, to see the effect of the default lighting context.



The added sphere will be using the **Standard shader** by default. Focusing the camera on the sphere will result on something like this:



Notice the reflection along the edges of the sphere as well as the subtle ambient, from brown (bottom) to the sky blue (top). By default, in an empty scene, all lighting context is derived from the skybox and a directional light (which is added to the Scene by default).

Of course this is the default setup, a single lighting and sky reflection may not be enough in some cases. You can easily add more lighting and reflection probes:

For an in-depth view on how reflection and light probes work please see the documentation on light probes and reflection probes.

Skyboxes

A Skybox, baked or procedural, can be an integral part of your lighting setup. It can be used to control the ambient lighting and the reflections in your objects in addition to **rendering** the sky. Procedural Skyboxes also allow you to set the colours directly and create a sun disc instead of using a bitmap - more information can be found in the Skybox Documentation

While reflecting the skybox can be useful for many objects in your scenes, particularly outdoor scenes, there are often cases where you need to vary the reflections an object uses - there may be dark areas in an outdoor scene, such as alleyways or dense forest, or you may have interior areas which require reflections to match each room.

To meet the needs of these various reflective requirements, Unity has reflection probes which allow you to sample the environment in your scene at a certain point in space, for use as the ambient light and reflection source for any objects near that point instead of the default skybox. Reflection probes can be placed around your scene in any locations where the scene's skybox is not sufficient or appropriate.

Global Illumination

The concept of Global Illumination is integral to Unity 5. Both the Standard **shader** and Unity 5's GI systems have been designed to play well with each other. The GI system takes care of creating and tracking bounced light, light from emissive materials and light from the environment. You can find the details here.

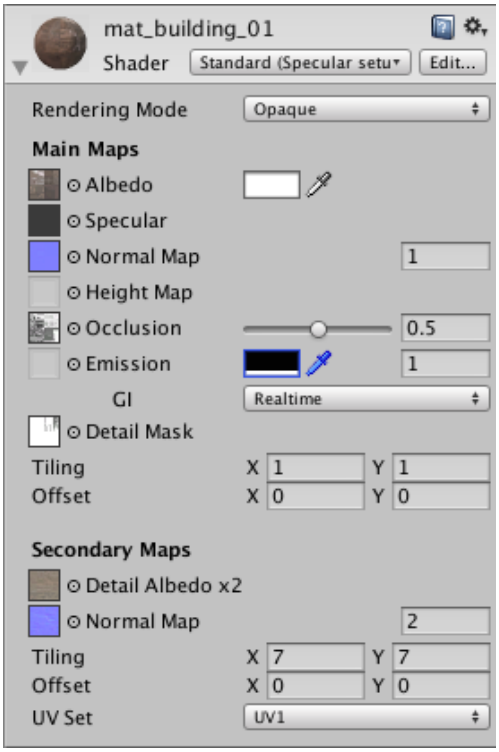
The context is a critical part of the overall look of the image. In this example you can see how a scene reflects changes in context, while content and camera remains the same.

The Content

The content is the term used to describe the objects in your scene that are being rendered. Their appearance is a result of the lighting context acting on the materials that have been applied to the objects.

The Material Editor

When viewing a material in the **inspector** which uses the Standard Shader, the editor displays all parameters for the material including textures, blending modes, masking and secondary maps. At a glance you be able to see which features are used and preview the material. As the Standard shader is data-driven: Unity will only use the shader code required by the configuration the user has set for the material. In other words if a feature or texture slot of the material is not used, there is no cost for it and the shader combination is optimised behind the scenes.



Tip: You can Ctrl+click on the texture thumbnails for a large preview, which will also let you check the contents of the color and alpha channels separately!

How to create a material

The Standard shader allows for many configurations in order to represent a great variety of material types. Values can be set with texture maps or colour pickers and sliders. Generally UV mapping is required in conjunction with textures to describe which part of your **mesh** refers to which part of the texture map. The Standard Shader material allows you therefore to have different material properties across the same **mesh** when used in conjunction with specular and smoothness map or a metallic map. In other words you can create rubber, metal and wood on one **mesh** where the resolution of the texture can exceed the polygon topology allowing for smooth borders and transition between material types, of course this has implications for a greater complexity in the workflow, but this will depend on your texture creation method.

Textures for your materials tend to be generated in one of two ways - painting and compositing in a 2D image editor like Photoshop, or rendering / baking from your 3D package, where you can also make use of higher resolution models to generate your normal maps and occlusion maps in addition to the albedo, specular and other maps. This workflow varies dependent on the external packages used.

Generally no texture map should contain inherent lighting (shadows, highlights, etc). One of the advantages of PBS is that objects react to light as you would expect, which is not possible if maps already contain lighting information.

Metallic vs Specular Workflow

Leave feedback

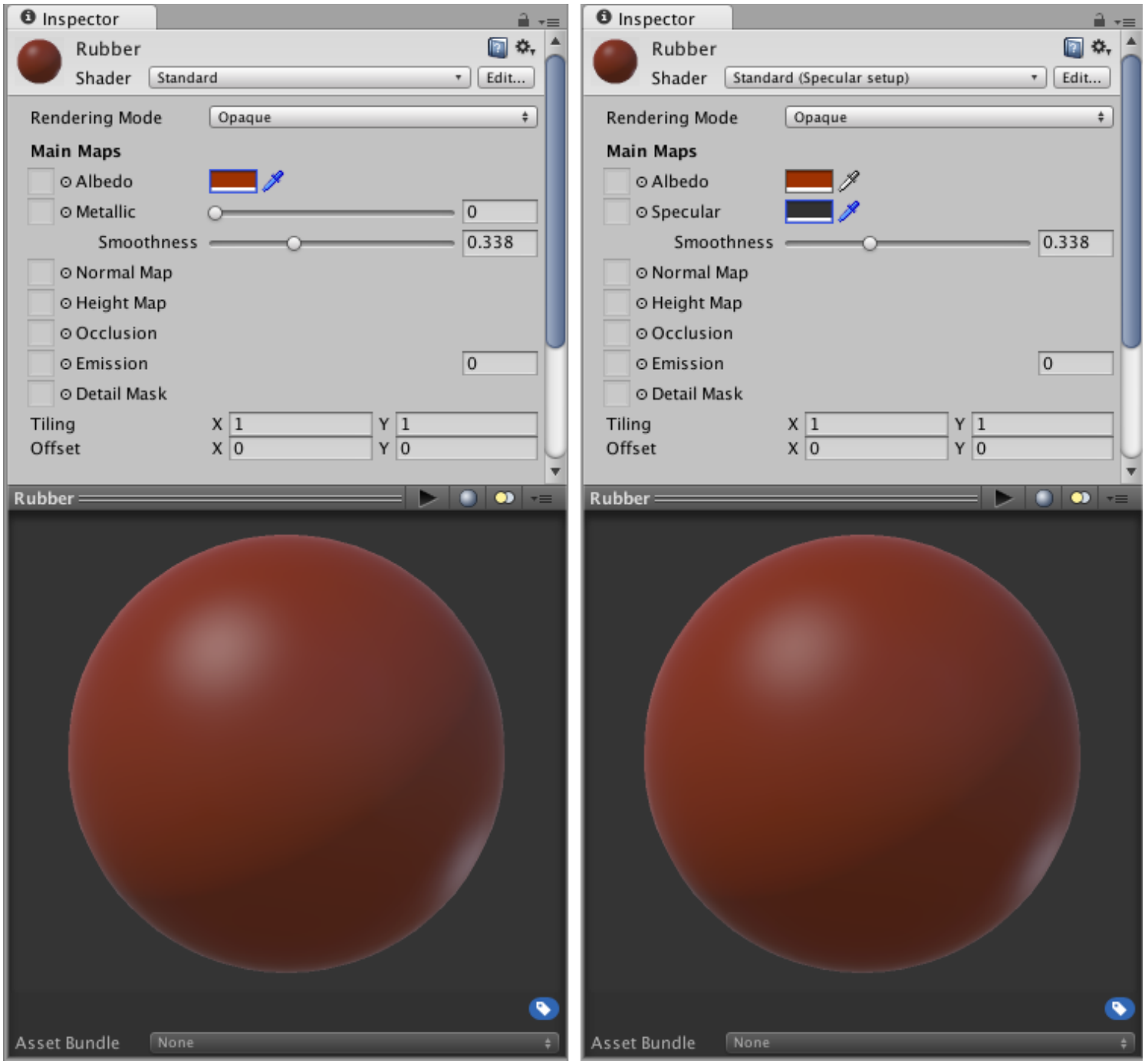
Two workflows

When creating a material using the **Standard shader** you will have the choice of using one of two flavours, “Standard” and “Standard (Specular setup)”. They differ in the data they take as follows:

Standard: The **shader** exposes a “metallic” value that states whether the material is metallic or not. In the case of a metallic material, the Albedo color will control the color of your specular reflection and most light will be reflected as specular reflections. Non metallic materials will have specular reflections that are the same color as the incoming light and will barely reflect when looking at the surface face-on.

Standard (Specular setup): Choose this shader for the classic approach. A Specular color is used to control the color and strength of specular reflections in the material. This makes it possible to have a specular reflection of a different color than the diffuse reflection for instance.

It is generally possible to achieve a good representation of most common material types using either method, so for the most part choosing one or the other is a matter of personal preference to suit your art workflow. For instance, to below is an example of a rubbery plastic material created in both Standard and Standard Specular workflows:



The fresnel effect visible at grazing angles in relation to the viewer is increasingly apparent as the surface of a material becomes smoother

The first image represents the metallic workflow, where we are setting this material to zero (non metallic). The second setup is nearly identical but we set the specular to nearly black (so we don't get metallic mirror-like reflections)

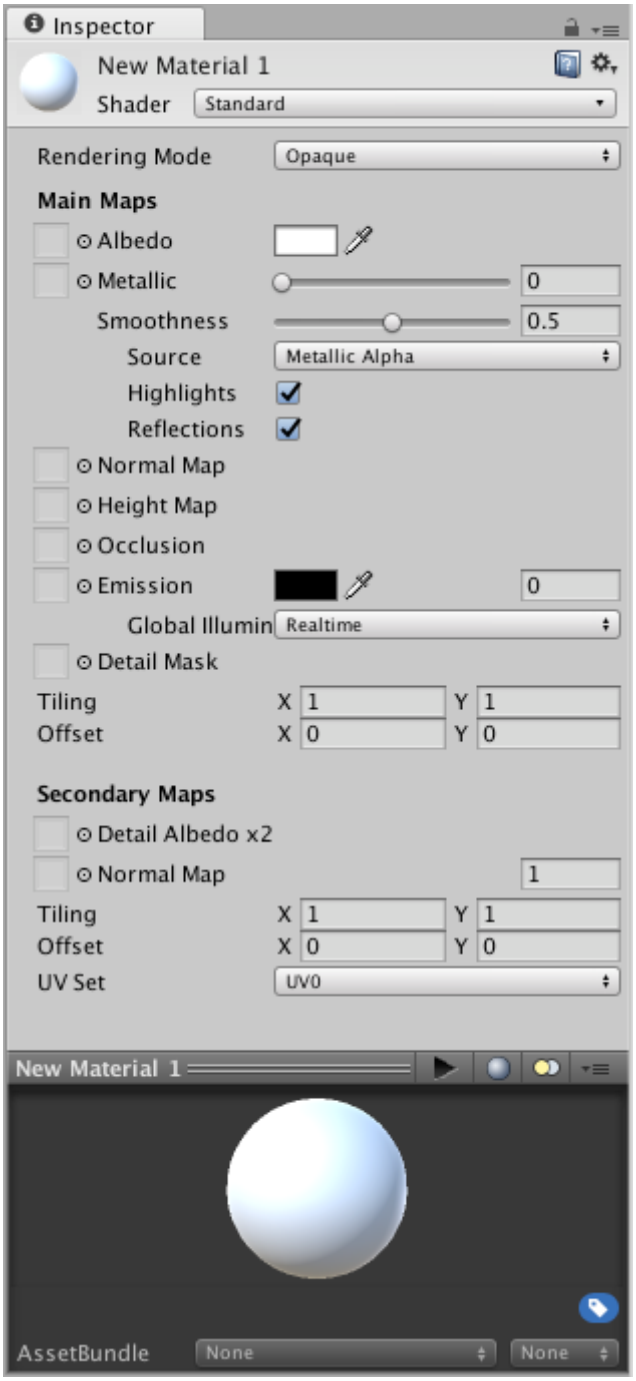
One might ask where do these values come from, what is "nearly black" and what makes grass different from aluminium exactly? In the world of **Physically Based Shading** we can use references from known real-world materials. Some of those references we have compiled into a handy set of charts you can use to create your materials.

Material parameters

Leave feedback

The **standard shader** presents you with a list of material parameters. These parameters vary slightly depending on whether you have chosen to work in the Metallic workflow mode or the Specular workflow mode. Most of the parameters are the same across both modes, and this page covers all the parameters for both modes.

These parameters can be used together to recreate the look of almost any real-world surface.



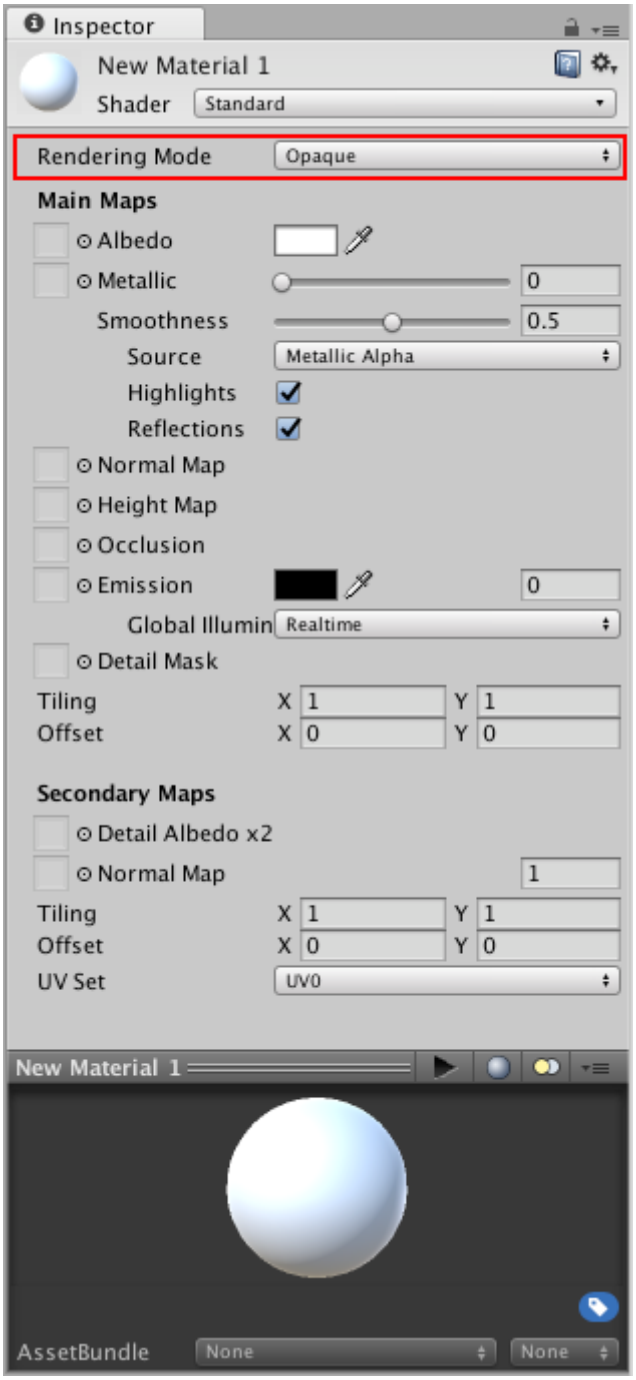
A Standard Shader material with default parameters and no values or textures assigned

Rendering Mode
Albedo Color & Transparency
Specular Mode: Specular
Metallic Mode: Metallic
Smoothness

Normal Map (Bump Mapping)
Height Map (Parallax Mapping)
Occlusion Map
Emission
Detail Mask & Maps
The Fresnel Effect

Rendering Mode

Leave feedback



A Standard Shader material with default parameters and no values or textures assigned. The Rendering Mode parameter is highlighted.

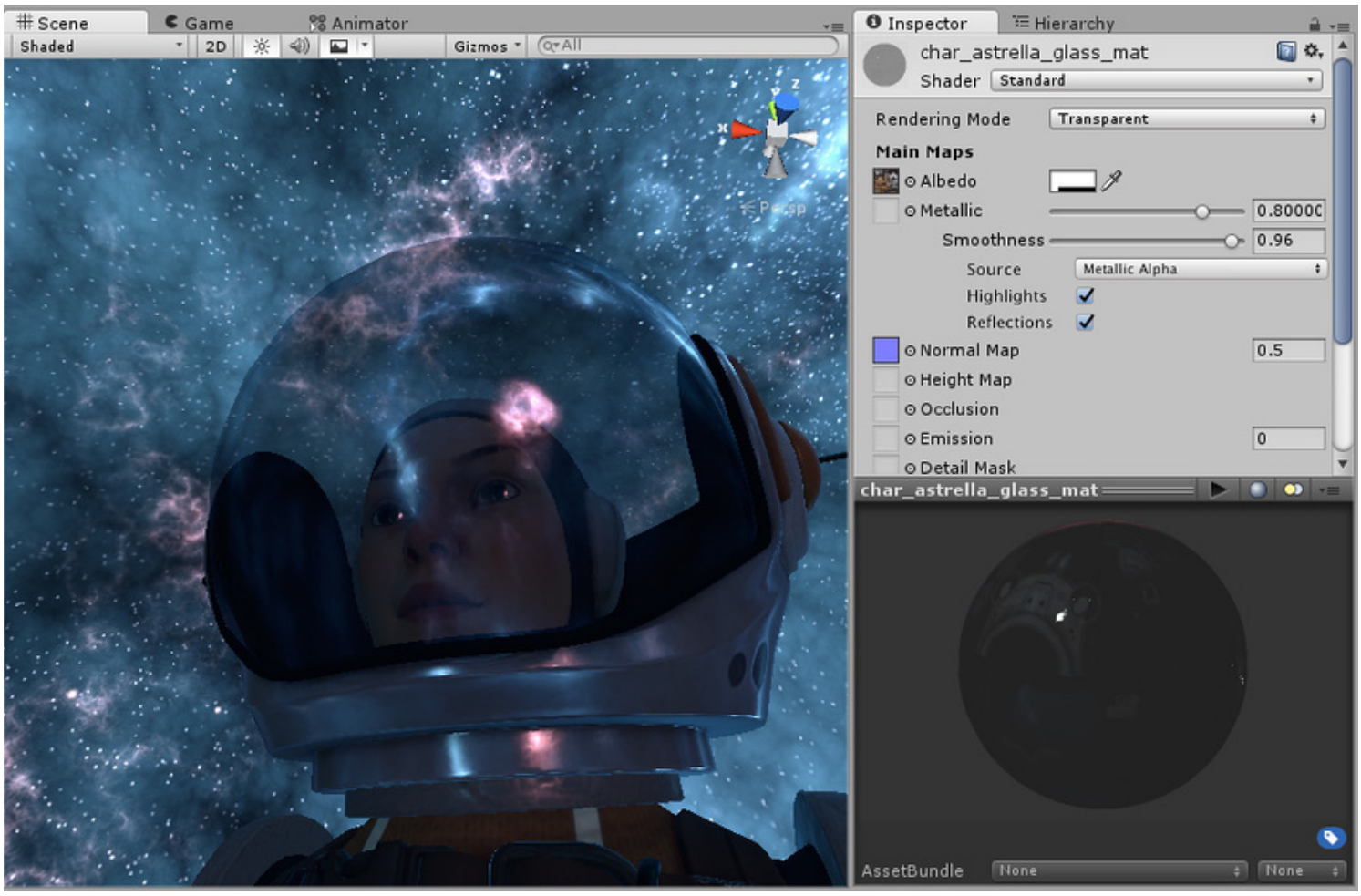
The first Material Parameter in the **Standard Shader** is **Rendering Mode**. This allows you to choose whether the object uses transparency, and if so, which type of blending mode to use.

Opaque - Is the default, and suitable for normal solid objects with no transparent areas.

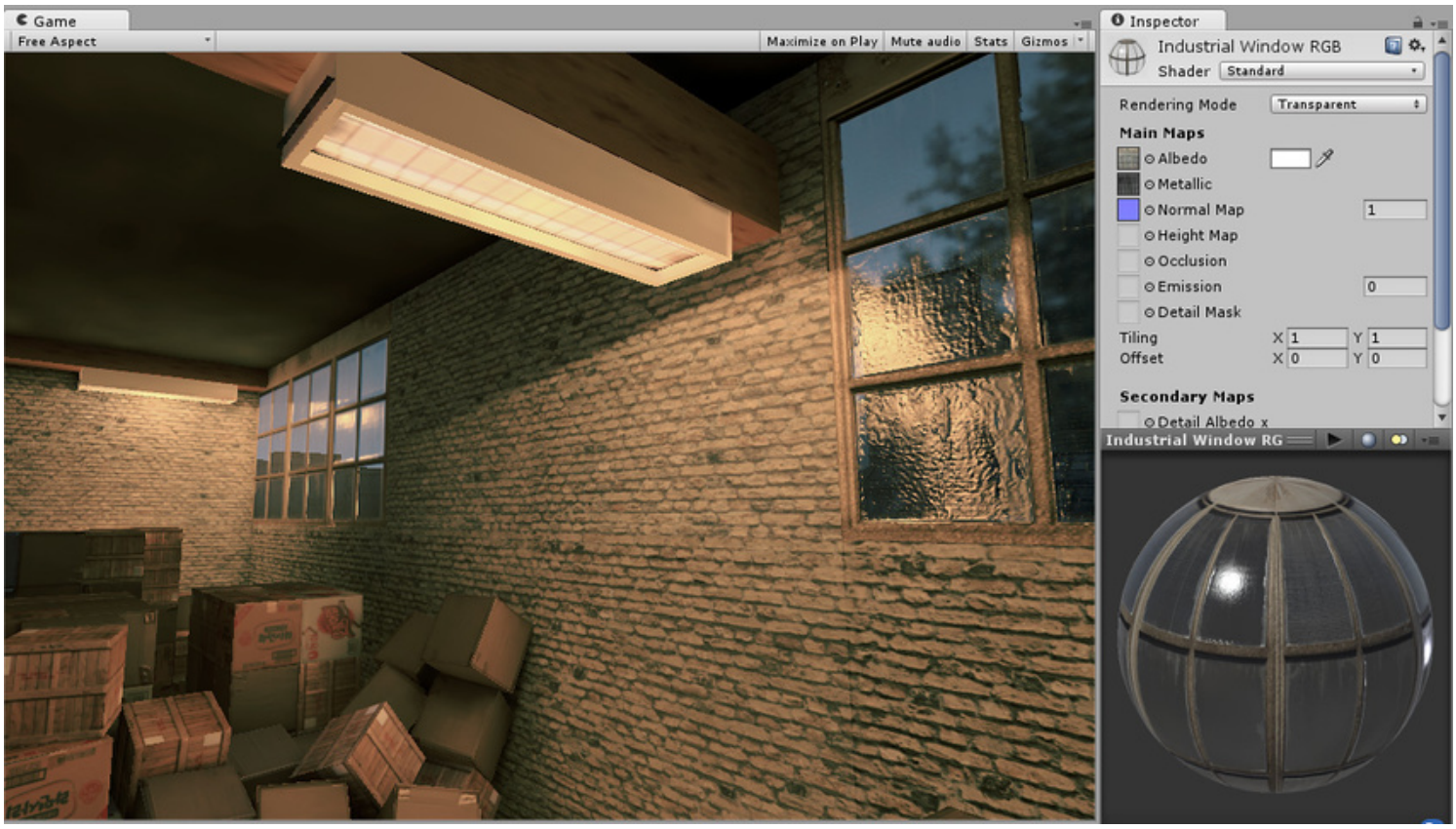
Cutout - Allows you to create a transparent effect that has hard edges between the opaque and transparent areas. In this mode, there are no semi-transparent areas, the texture is either 100% opaque, or invisible. This is useful when using transparency to create the shape of materials such as leaves, or cloth with holes and tatters.

Transparent - Suitable for rendering realistic transparent materials such as clear plastic or glass. In this mode, the material itself will take on transparency values (based on the texture's alpha channel and the alpha of the tint colour), however reflections and lighting highlights will remain visible at full clarity as is the case with real transparent materials.

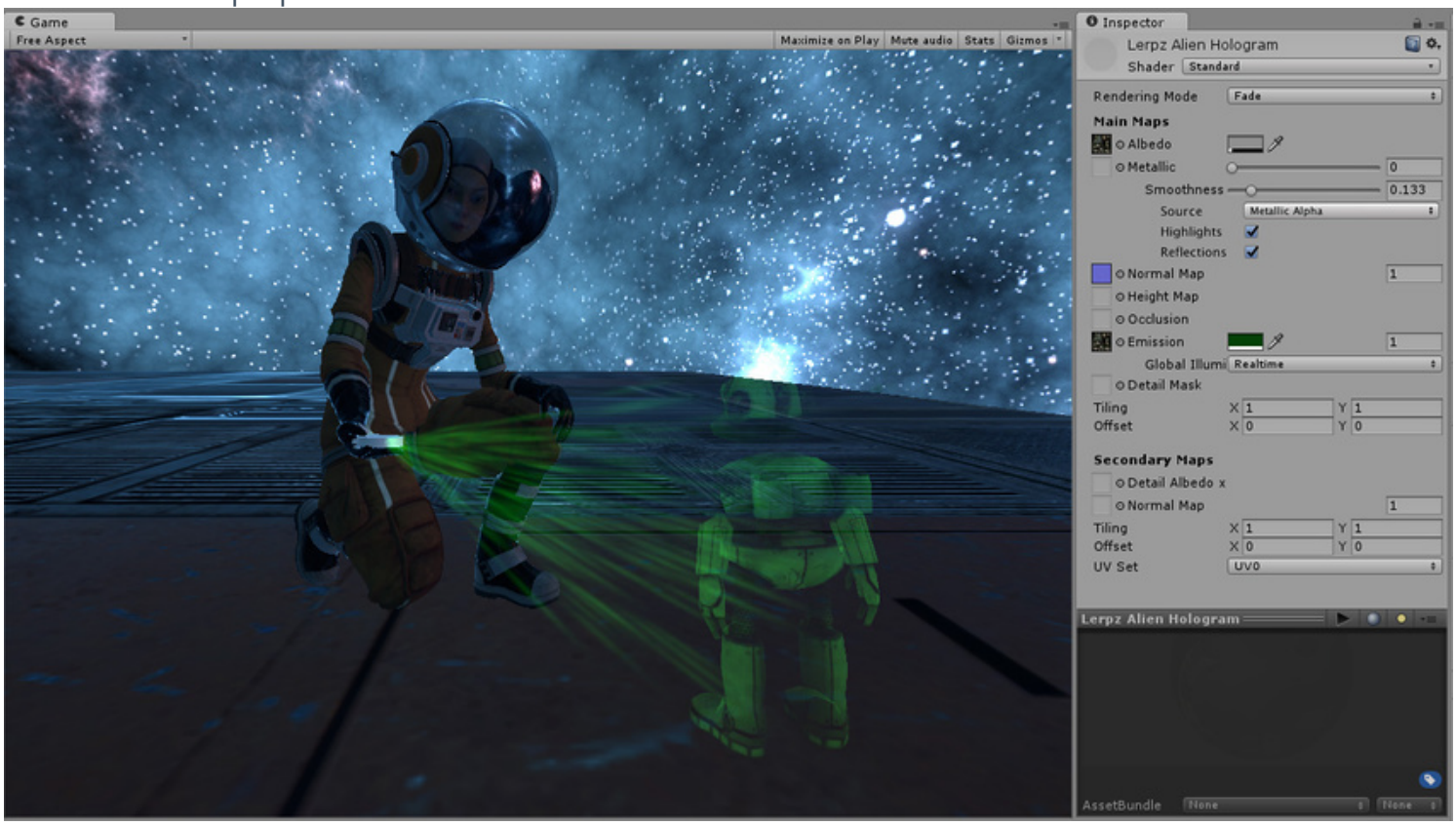
Fade - Allows the transparency values to entirely fade an object out, including any specular highlights or reflections it may have. This mode is useful if you want to animate an object fading in or out. It is not suitable for rendering realistic transparent materials such as clear plastic or glass because the reflections and highlights will also be faded out.



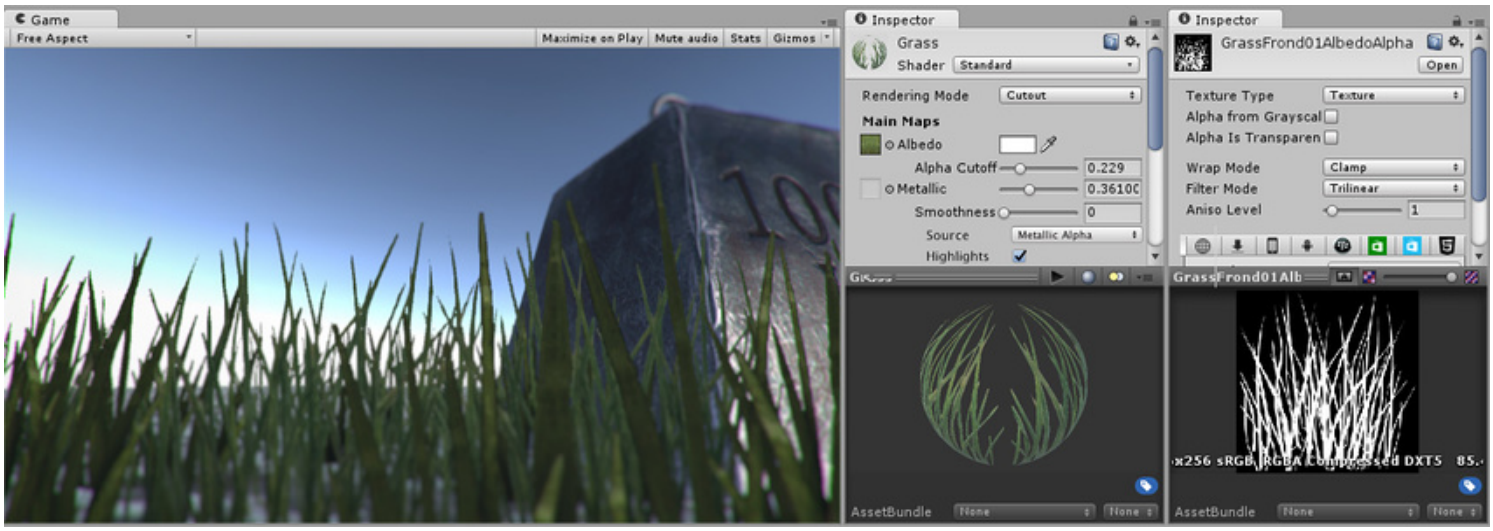
The helmet visor in this image is rendered using the Transparent mode, because it is supposed to represent a real physical object that has transparent properties. Here the visor is reflecting the skybox in the scene.



These windows use Transparent mode, but have some fully opaque areas defined in the texture (the window frames). The Specular reflection from the light sources reflects off the transparent areas and the opaque areas.



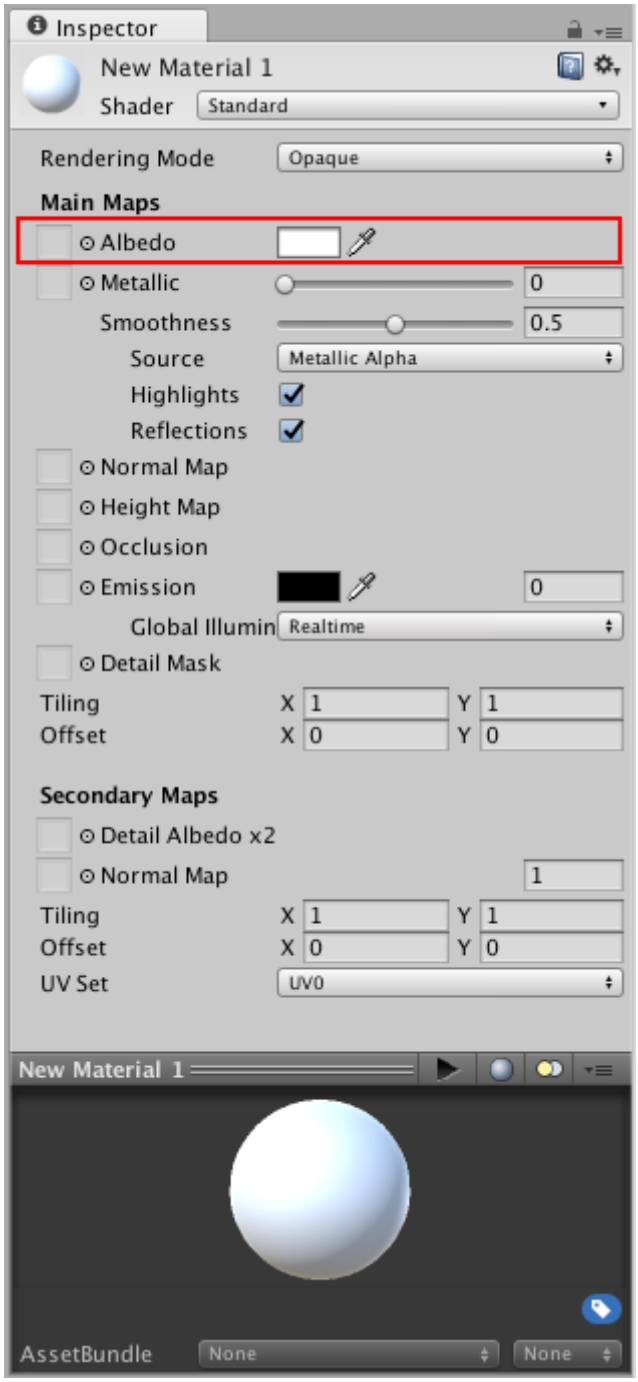
The hologram in this image is rendered using the Fade mode, because it is supposed to represent an opaque object that is partially faded out.



The grass in this image is rendered using the Cutout mode. This gives clear sharp edges to objects which is defined by specifying a cut-off threshold. All parts of the image with the alpha value above this threshold are 100% opaque, and all parts below the threshold are invisible. To the right in the image you can see the material settings and the alpha channel of the texture used.

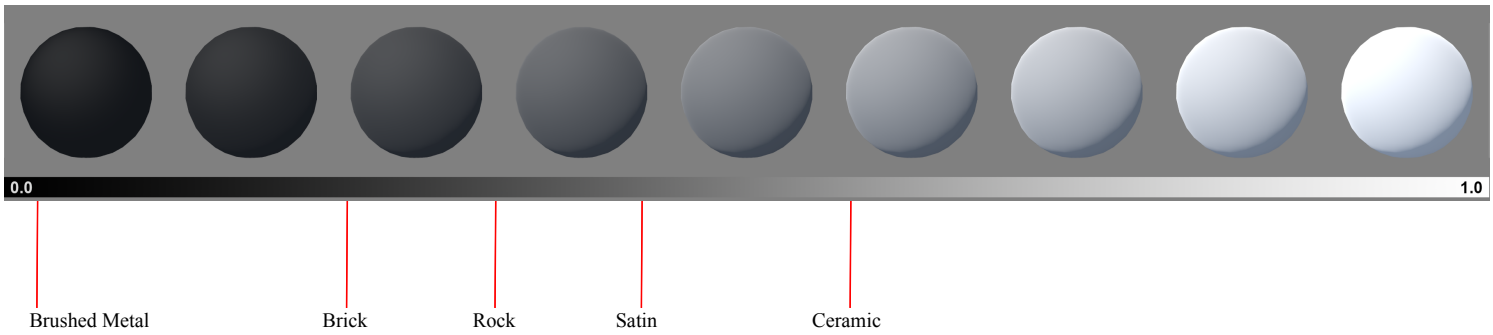
Albedo Color and Transparency

Leave feedback



A Standard Shader material with default parameters and no values or textures assigned. The Albedo Color parameter is highlighted.

The Albedo parameter controls the base color of the surface.



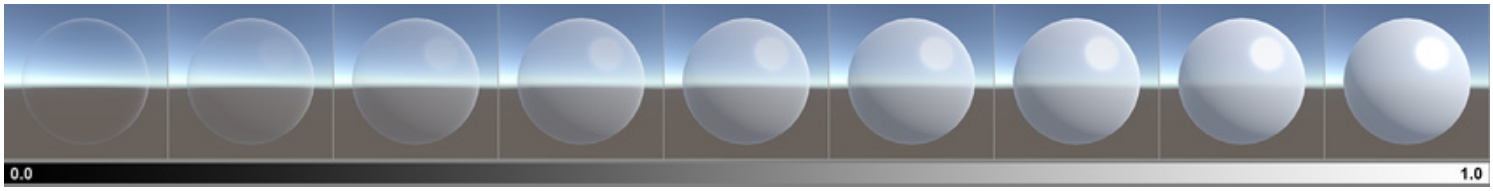
A range of black to white albedo values

Specifying a single color for the Albedo value is sometimes useful, but it is far more common to assign a texture map for the Albedo parameter. This should represent the colors of the surface of the object. It's important to note that the Albedo texture should **not** contain any lighting, since the lighting will be added to it based on the context in which the object is seen.

Two examples of typical Albedo texture maps. On the left is a texture map for a character model, and on the right is a wooden crate. Notice there are no shadows or lighting highlights.

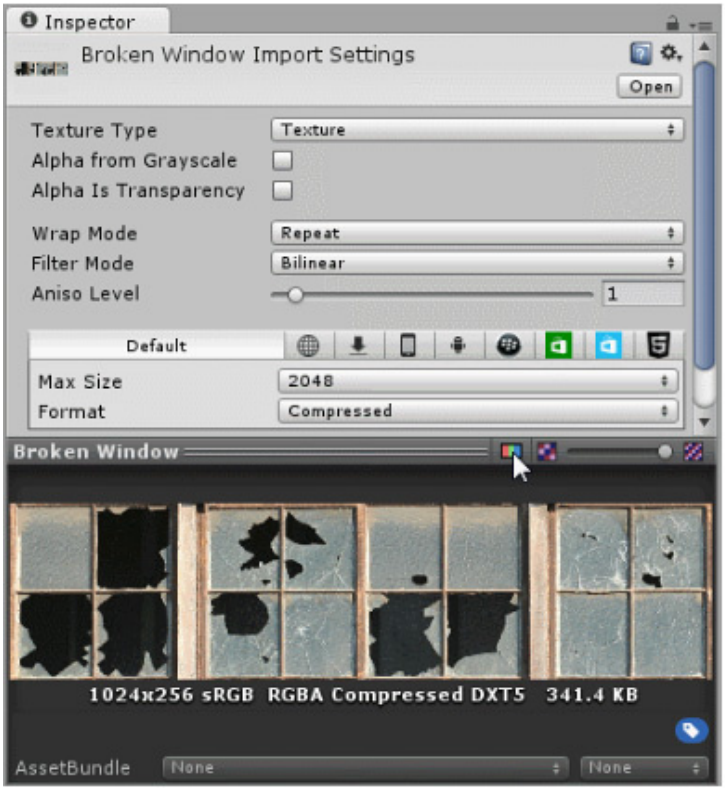
Transparency

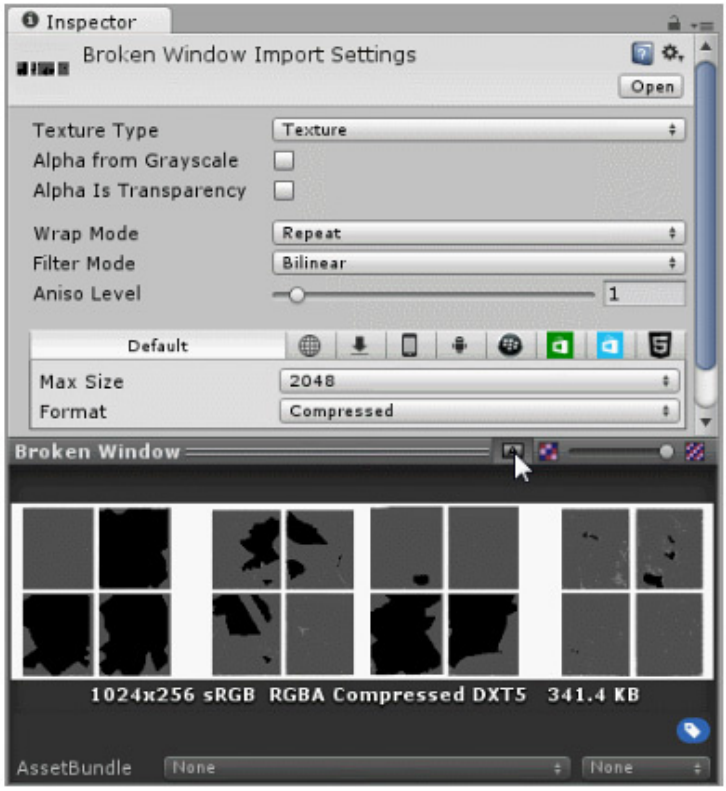
The alpha value of the Albedo colour controls the transparency level for the material. This only has an effect if the **Rendering** Mode for the material is set to one of the transparent mode, and not **Opaque**. As mentioned above, picking the correct transparency mode is important because it determines whether or not you will still see reflections and specular highlights at full value, or whether they will be faded out according to the transparency values too.



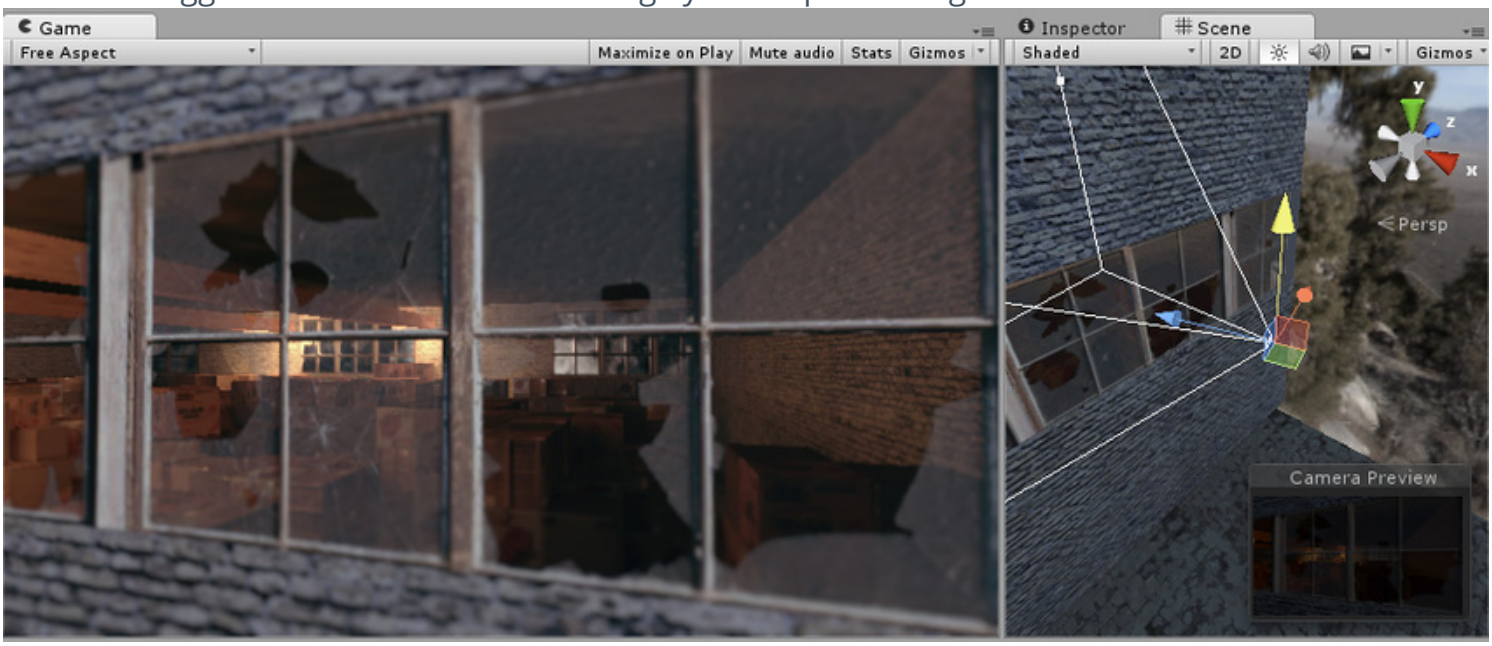
A range of transparency values from 0 to 1, using the Transparent mode suitable for realistic transparent objects

When using a texture assigned for the Albedo parameter, you can control the transparency of the material by ensuring your albedo texture image has an **alpha channel**. The alpha channel values are mapped to the transparency levels with white being fully opaque, and black being fully transparent. This will have the effect that your material can have areas of varying transparency.





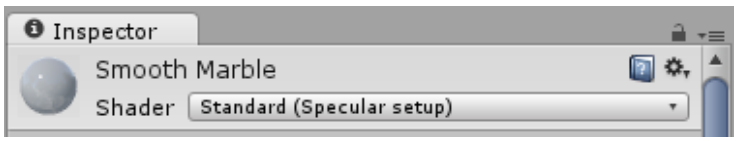
An imported texture with RGB channels and an Alpha Channel. You can click the RGB/A button as shown to toggle which channels of the image you are previewing.



The end result, peering through a broken window into a building. The gaps in the glass are totally transparent, while the glass shards are partially transparent and the frame is fully opaque.

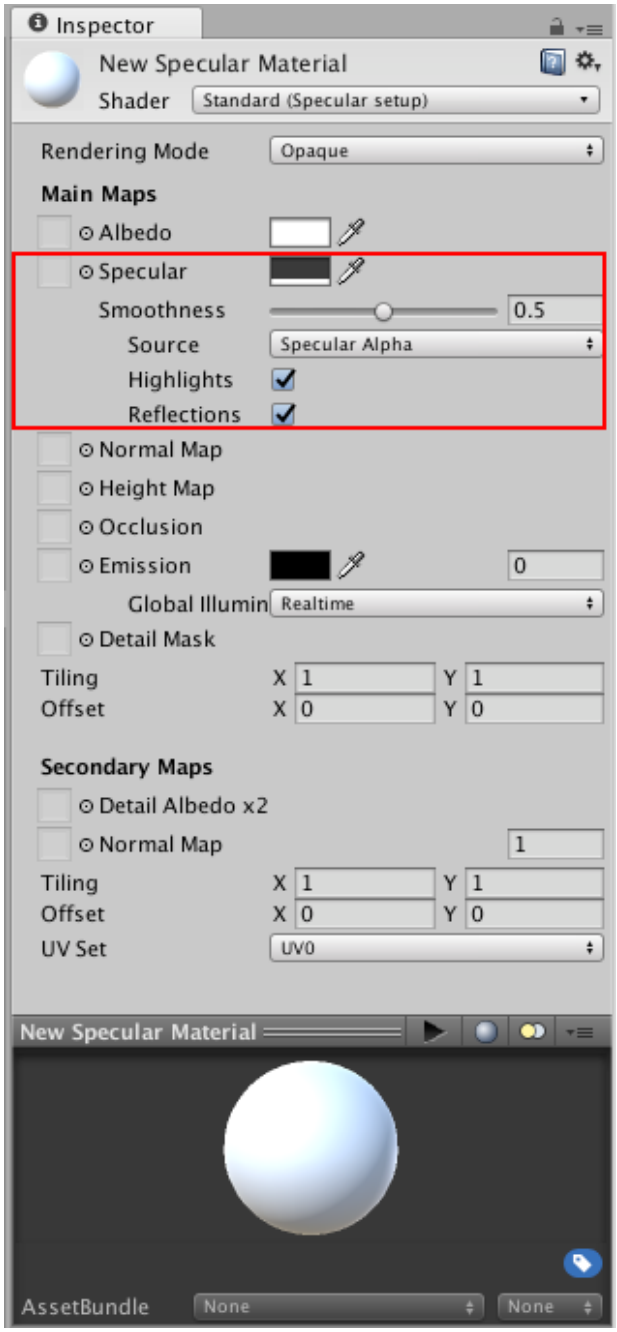
Specular mode: Specular parameter

Leave feedback



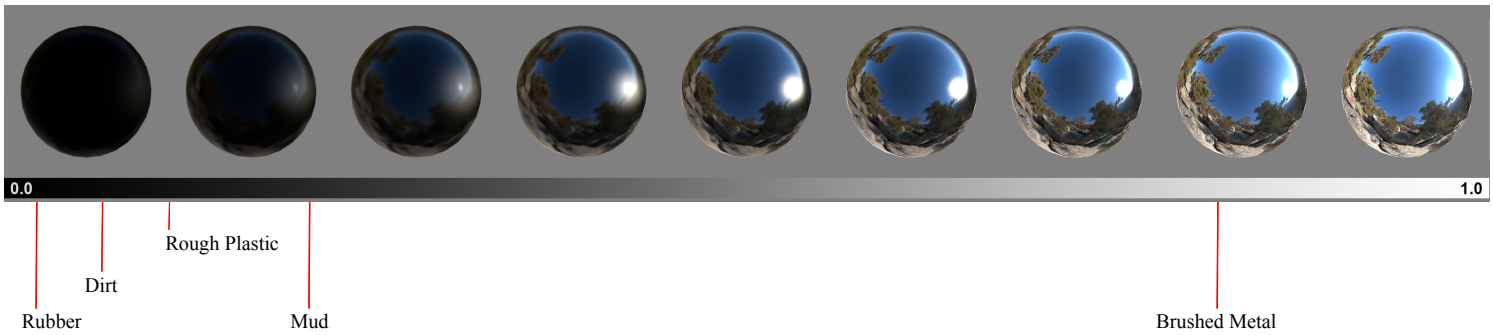
Specular parameter

The **Specular** parameter is only visible when using the **Specular setup**, as shown in the **Shader** field in the image above. Specular effects are essentially the direct reflections of light sources in your **Scene**, which typically show up as bright highlights and shine on the surface of objects (although specular highlights can be subtle or diffuse too).



Both the **Specular setup** and **Metallic setup** produce specular highlights, so the choice of which to use is more a matter of setup and your artistic preference. In the **Specular setup** you have direct control over the brightness and tint colour of specular highlights, while in the **Metallic setup** you control other parameters and the intensity and colour of the specular highlights emerge as a natural result of the other parameter settings.

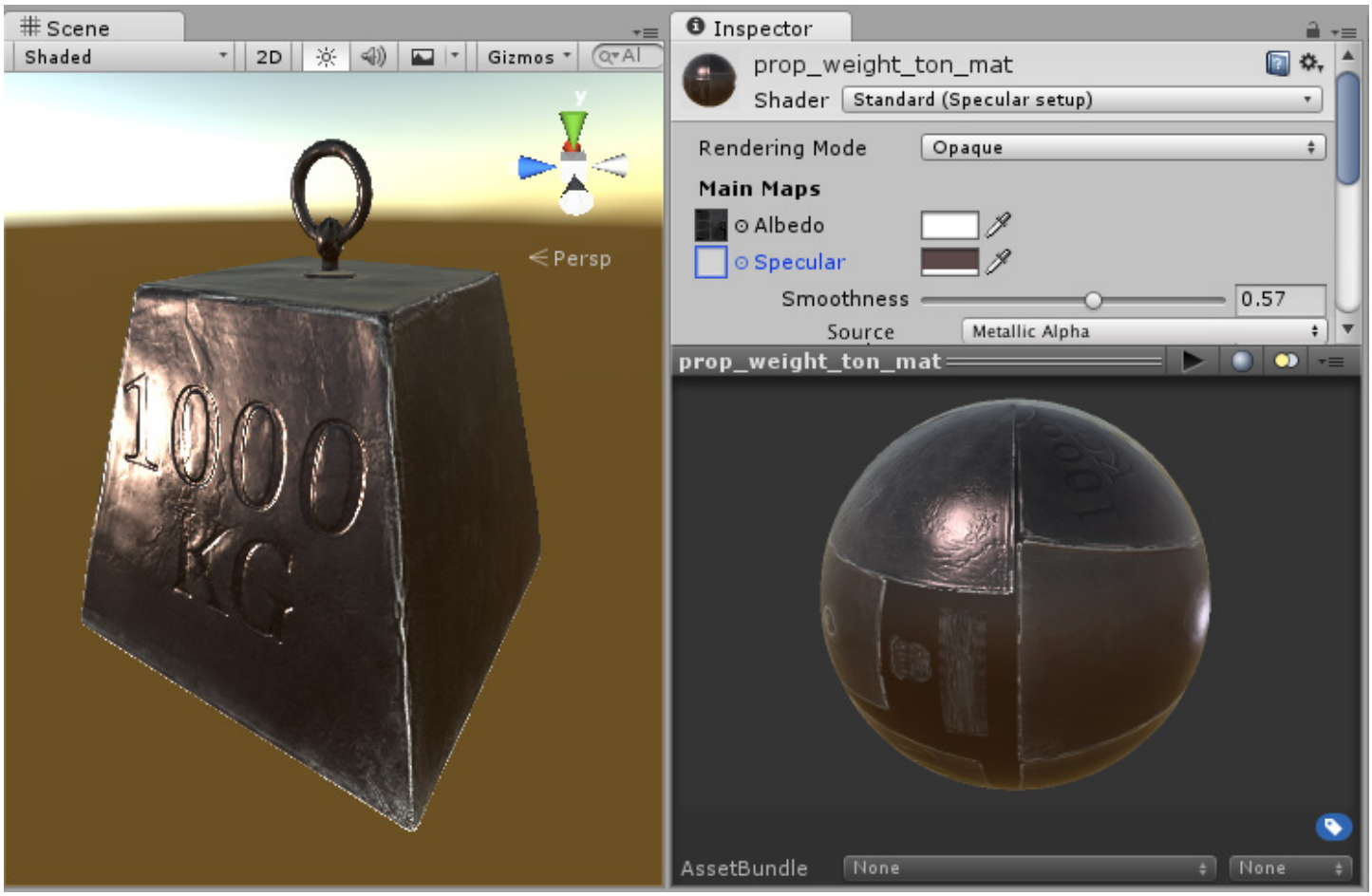
When working in Specular mode, the RGB colour in the **Specular** parameter controls the strength and colour tint of the specular reflectivity. This includes shine from light sources and reflections from the environment. The Smoothness parameter controls the clarity of the specular effect. With a low smoothness value, even strong specular reflections appear blurred and diffuse. With a high smoothness value, specular reflections are crisper and clearer.



The Specular Smoothness values from 0 to 1

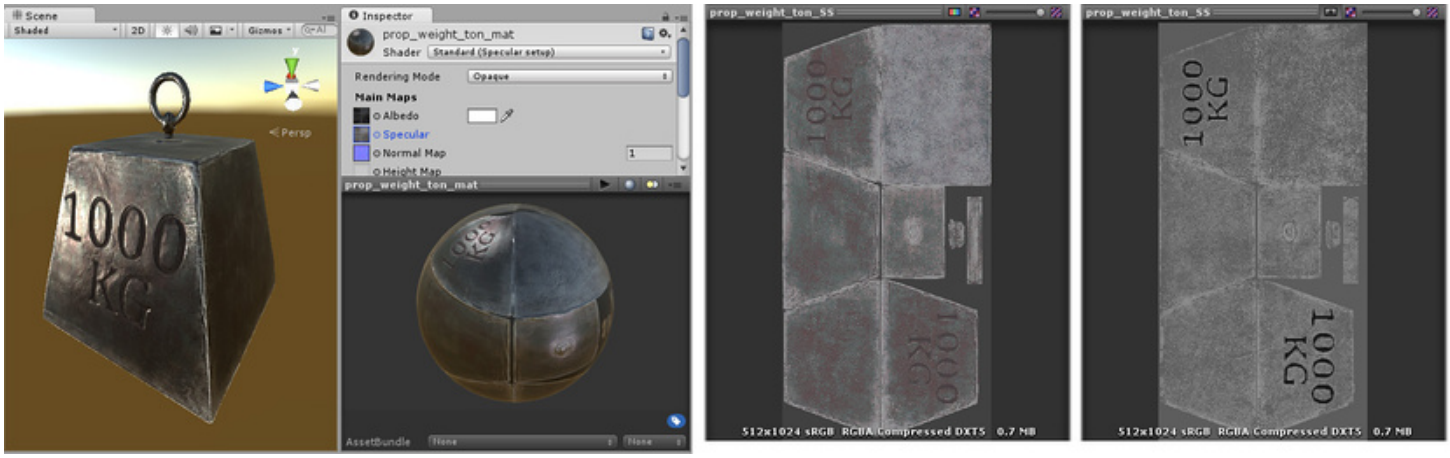
You might want to vary the **Specular** values across the surface of your material - for example, if your Texture contains a character's coat that has some shiny buttons. You would want the buttons to have a higher specular value than the fabric of the clothes. To achieve this, assign a Texture map instead of using a single slider value. This allows greater control over the the strength and colour of the specular light reflections across the surface of the material, according to the **pixel** colours of your specular map.

When a Texture is assigned to the **Specular** parameter, both the **Specular** parameter and **Smoothness** slider disappear. Instead, the specular levels for the material are controlled by the values in the **Red**, **Green** and **Blue** channels of the Texture itself, and the Smoothness levels for the material are controlled by the Alpha channel of the same Texture. This provides a single Texture which defines areas as being rough or smooth, and have varying levels and colors of specularity. This is very useful when working Texture maps that cover many areas of a model with varying requirements; for example, a single character Texture map often includes multiple surface requirements such as leather shoes, fabric of the clothes, skin for the hands and face, and metal buckles.



An example of a 1000kg weight with a strong specular reflection from a directional light.

Here, the specular reflection and smoothness are defined by a colour and the **Smoothness** slider. No Texture has been assigned, so the specular and smoothness level is constant across the whole surface. This is not always desirable, particularly in the case where your Albedo Texture maps to a variety of different areas on your model (also known as a Texture atlas).



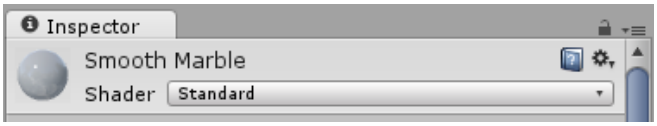
The same model, but with a specular map assigned, instead of using a constant value.

Here, a Texture map controls the specularity and smoothness. This allows the specularity to vary across the surface of the model. Notice the edges have a higher specular effect than the centre, there are some subtle colour responses to the light, and the area inside the lettering no longer has specular highlights. Pictured to the right are the RGB channels controlling the specular colour and strength, and the Alpha channel controlling the smoothness.

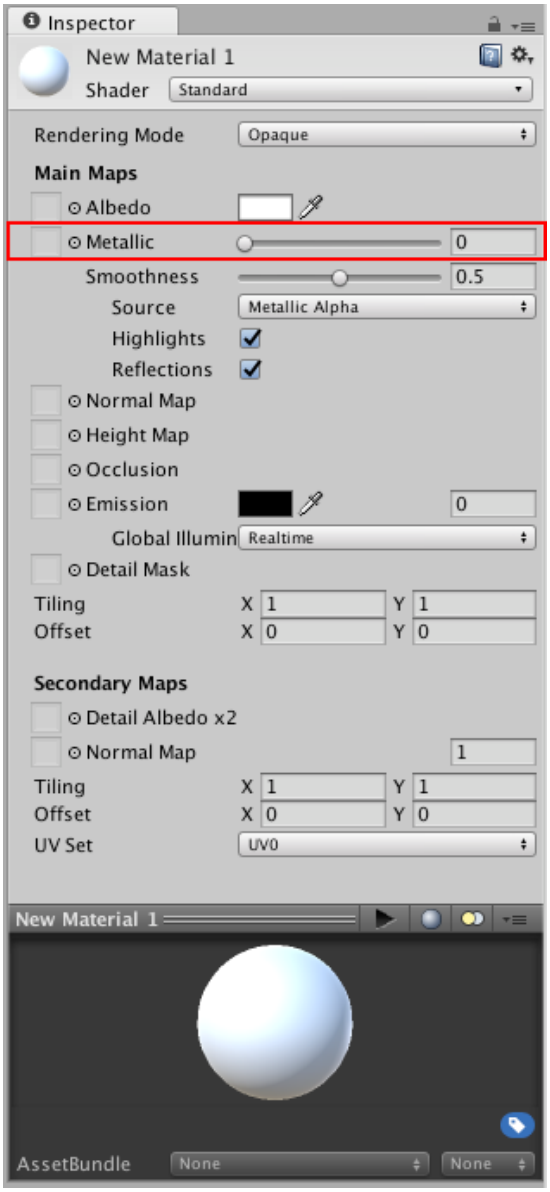
Note: A black specular color (0,0,0) results in nulling out the specular effect.

Metallic mode: Metallic Parameter

Leave feedback



When working in the **Metallic workflow** (as opposed to the Specular workflow), the the reflectivity and light response of the surface are modified by the Metallic level and the Smoothness level.

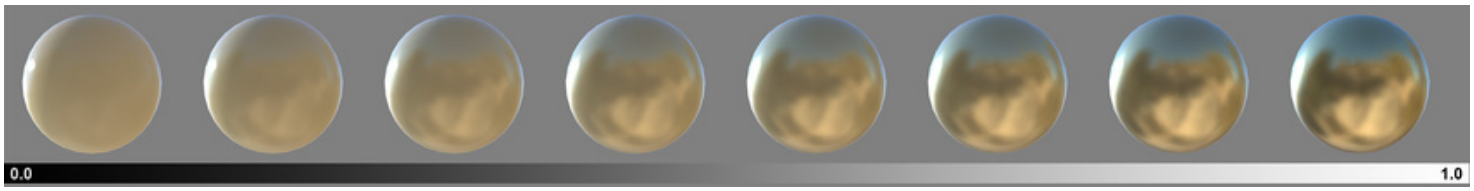


Specular reflections are still generated when using this workflow but they arise naturally depending on the settings you give for the Metallic and Smoothness levels, rather than being explicitly defined.

Metallic mode is not just for materials which are supposed to look metallic! This mode is known as metallic because of the way you have control over how metallic or non-metallic a surface is.

Metallic parameter

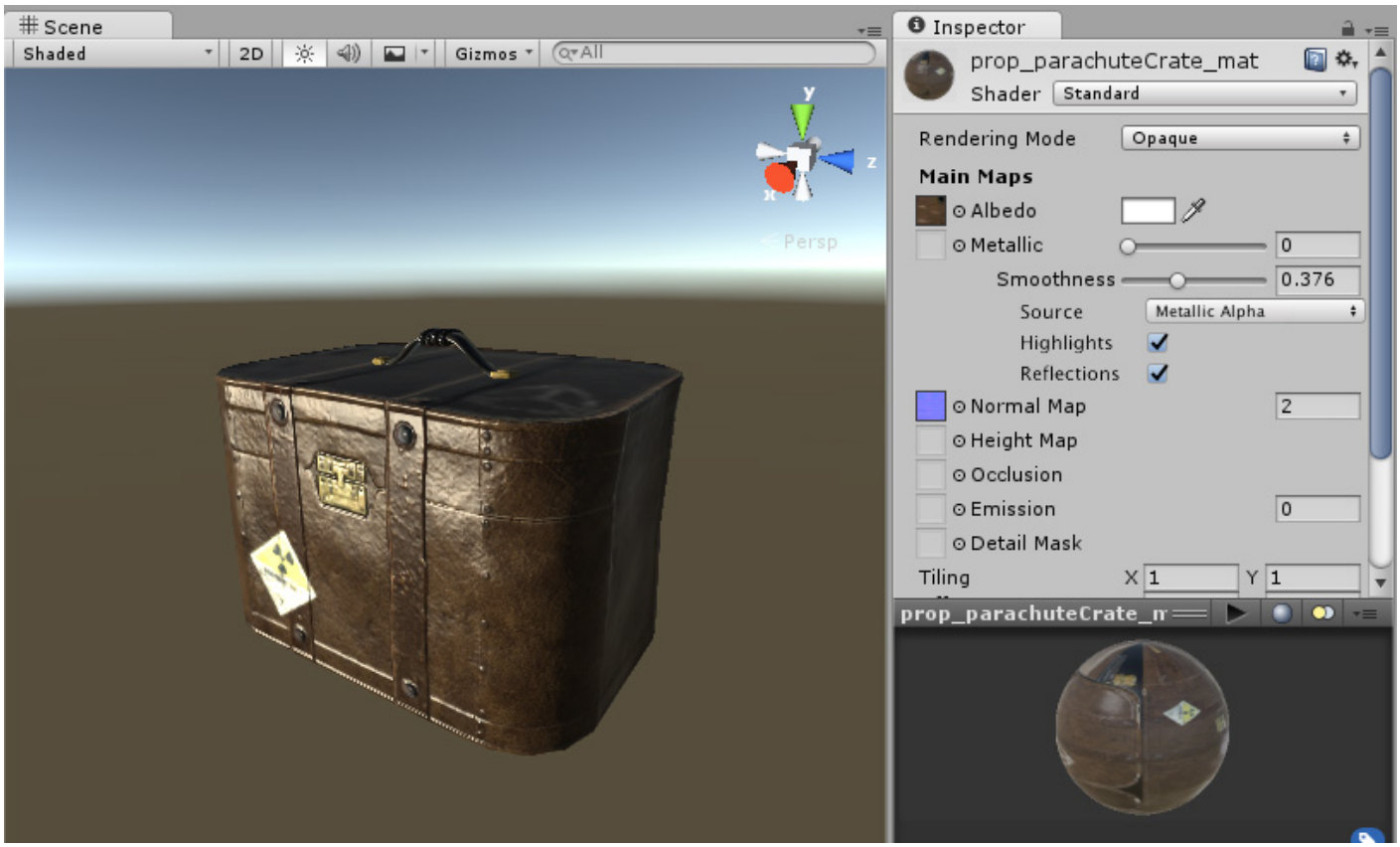
The metallic parameter of a material determines how “metal-like” the surface is. When a surface is more metallic, it reflects the environment more and its albedo colour becomes less visible. At full metallic level, the surface colour is entirely driven by reflections from the environment. When a surface is less metallic, its albedo colour is more clear and any surface reflections are visible on top of the surface colour, rather than obscuring it.



A range of metallic values from 0 to 1 (with smoothness at a constant 0.8 for all samples)

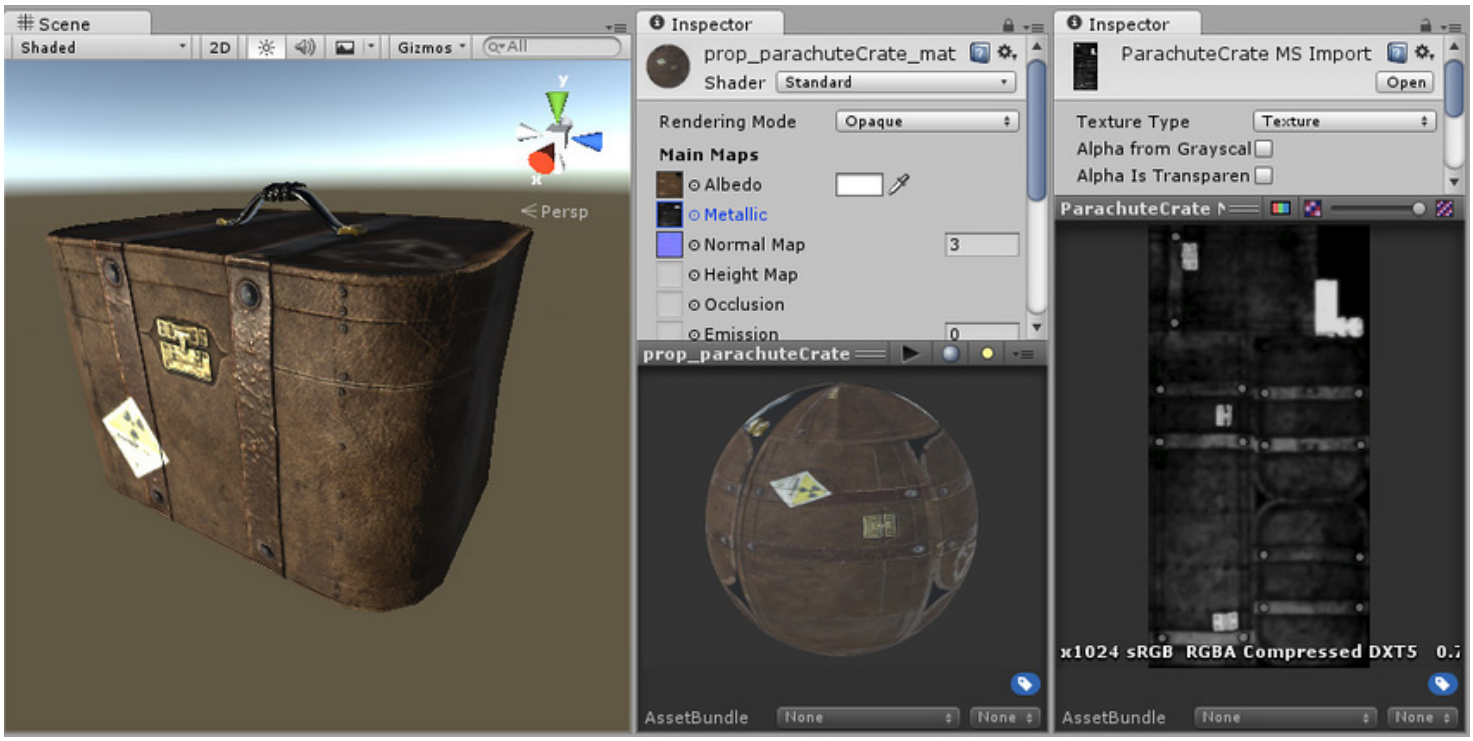
By default, with no texture assigned, the Metallic and Smoothness parameters are controlled by a slider each. This is enough for some materials. However if your model's surface has areas with a mixture of surface types in the albedo texture, you can use a texture map to control how the metallic and smoothness levels vary across the surface of the material. For instance if your texture contains a character's clothing including some metal buckles and zips. You would want the buckles and zips to have a higher metallic value than the fabric of the clothes. To achieve this, instead of using a single slider value, a texture map can be assigned which contains lighter **pixel** colours in the areas of the buckles and zips, and darker values for the fabric.

With a texture assigned to the Metallic parameter, both the Metallic and Smoothness sliders will disappear. Instead, the Metallic levels for the material are controlled by the values in the Red channel of the texture, and the Smoothness levels for the material are controlled by the Alpha channel of the texture. (This means the Green and Blue channels are ignored). This means you have a single texture which can define areas as being rough or smooth, and metallic or non-metallic, which is very useful when working texture maps that cover many areas of a model with varying requirements - for example a single character texture map often includes multiple surface requirements - leather shoes, cloth clothes, skin for the hands and face and metal buckles.



This image shows a case model with no metallic map

In the example above, the case has an albedo map, but no texture for Metallic. This means the whole object has a single metallic and smoothness value, which is not ideal. The leather straps, the metal buckles, the sticker and the handle should all appear to have different surface properties.

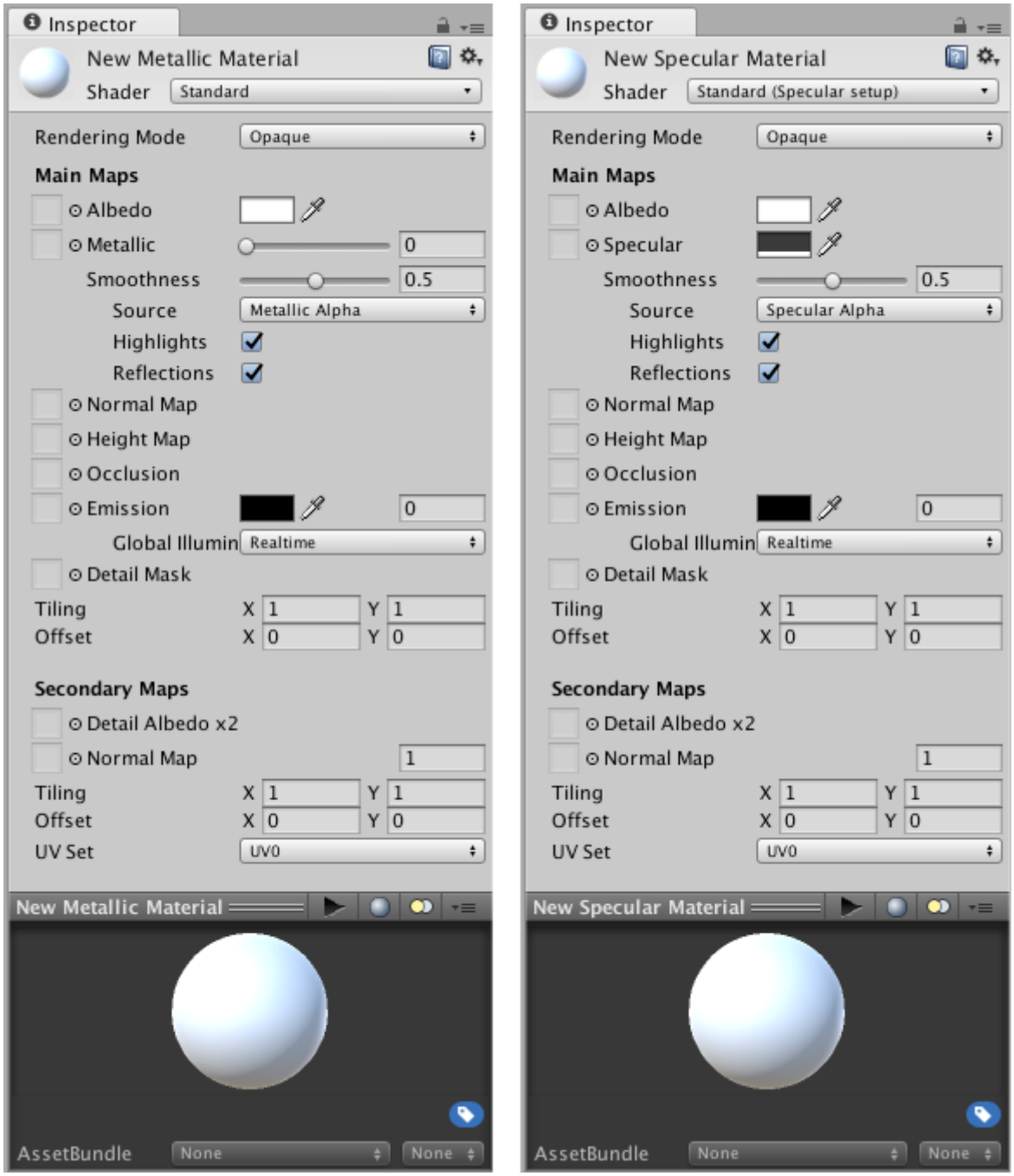


This image shows a case model with a metallic map applied

In this example, a Metal/Smoothness texture map has been assigned. The buckle now has a high metallic value and responds to light accordingly. The leather straps are shinier than the leather body of the box, however they have a low “Metallic” value, so it appears to be shiny non-metal surface. The black and white map on the far right shows the lighter areas for metal, and mid to low greys for the leather.

Smoothness

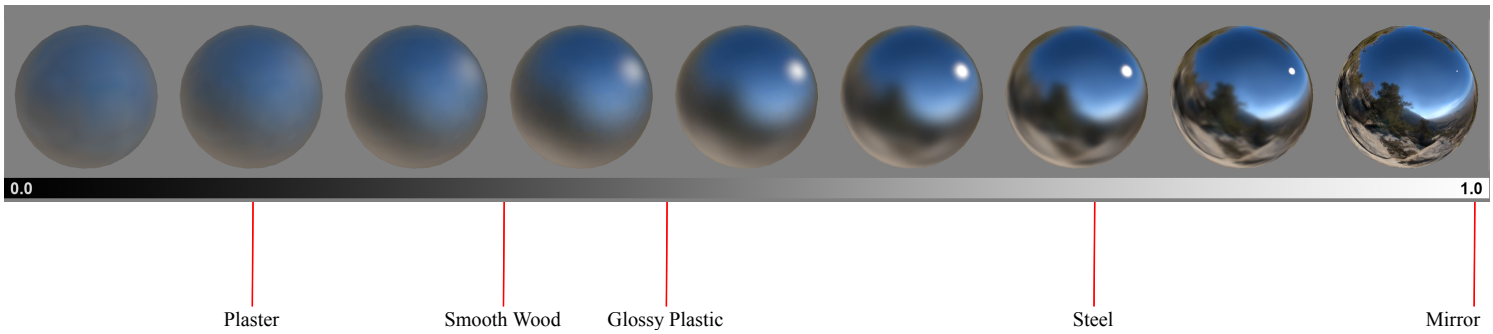
Leave feedback



The smoothness parameter, shown in both Metallic & Specular shader modes.

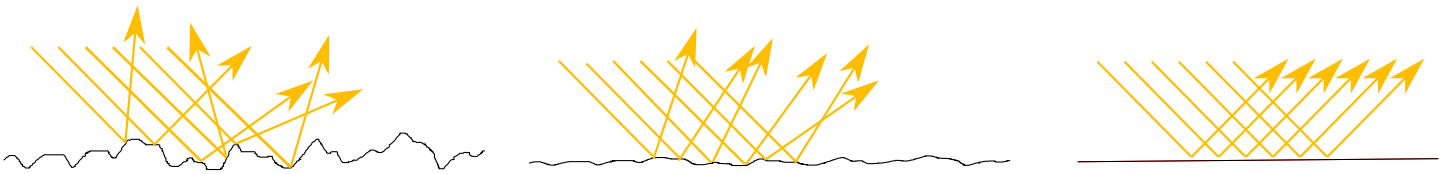
The concept of Smoothness applies to both the Specular workflow and the Metallic workflow, and works in very much the same way in both. By default, without a Metallic or Specular texture map assigned, the smoothness of the material is controlled by a slider. This slider allows you to control the “microsurface detail” or smoothness across a surface.

Both **shader** modes are shown above, because if you choose to use a texture map for the **Metallic** or **Specular** parameter, the smoothness values are taken from that same map. This is explained in further detail down the page.



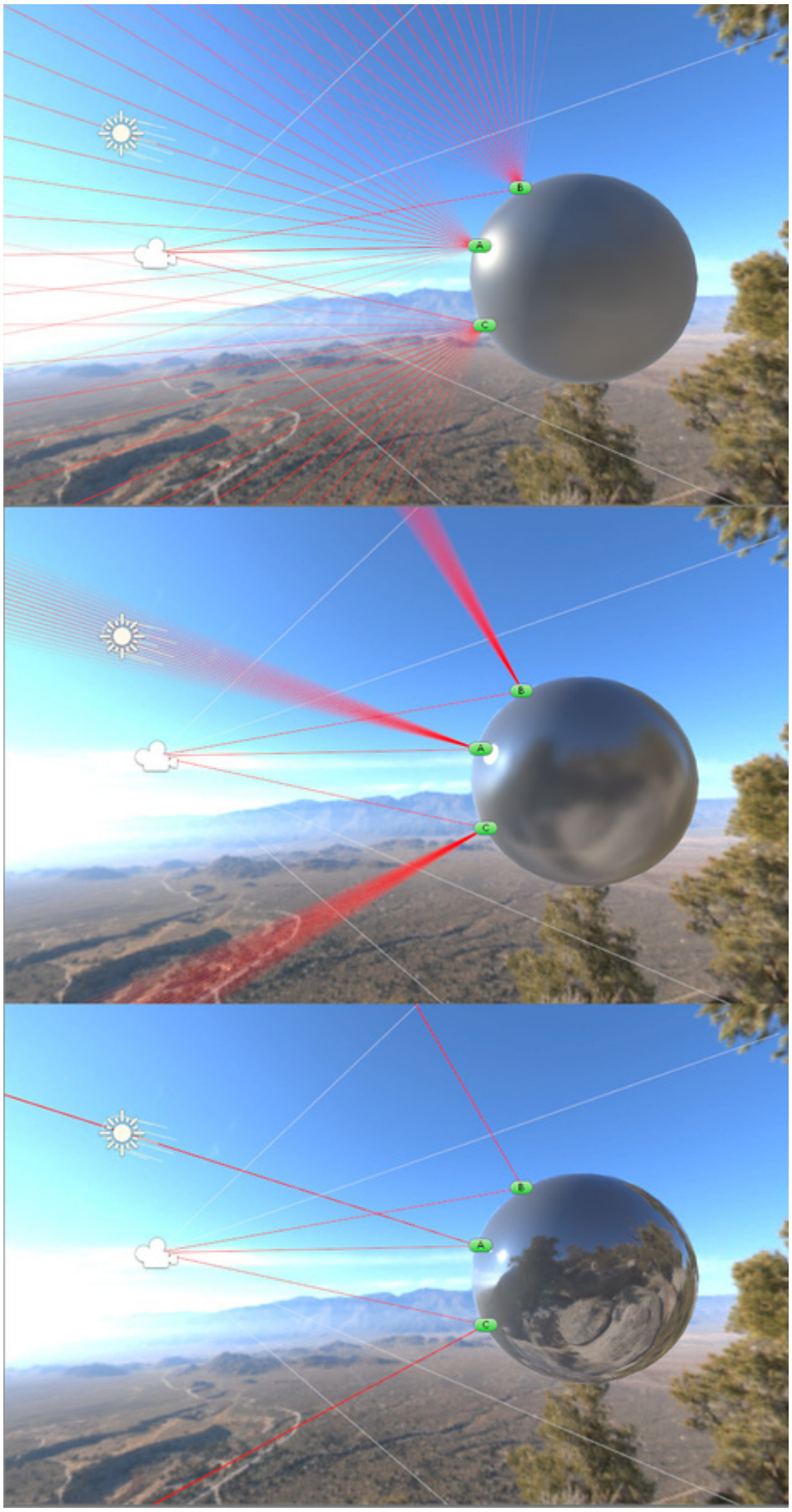
A range of smoothness values from 0 to 1

The “microsurface detail” is not something directly visible in Unity. It is a concept used in the lighting calculations. You can, however, see the effect of this microsurface detail represented by the amount the light that is diffused as it bounces off the object. With a smooth surface, all light rays tend to bounce off at predictable and consistent angles. Taken to its extreme, a perfectly smooth surface reflects light like a mirror. Less smooth surfaces reflect light over a wider range of angles (as the light hits the bumps in the microsurface), and therefore the reflections have less detail and are spread across the surface in a more diffuse way.



A comparison of low, medium and high values for smoothness (left to right), as a diagram of the theoretical microsurface detail of a material. The yellow lines represent light rays hitting the surface and reflecting off the angles encountered at varying levels of smoothness.

A smooth surface has very low microsurface detail, or none at all, so light bounces off in uniform ways, creating clear reflections. A rough surface has high peaks and troughs in its microsurface detail, so light bounces off in a wide range of angles which, when averaged out, create a diffuse colour with no clear reflections.



A comparison of low, medium and high values for smoothness (top to bottom).

At low smoothness levels, the reflected light at each point on the surface comes from a wide area, because the microsurface detail is bumpy and scatters light. At high values of smoothness, the light at each point comes from

a narrowly focused area, giving a much clearer reflection of the object's environment.

Using a Smoothness Texture Map

In a similar way to many of the other parameters, you can assign a texture map instead of using a single slider value. This allows you greater control over the strength and colour of the specular light reflections across the surface of the material.

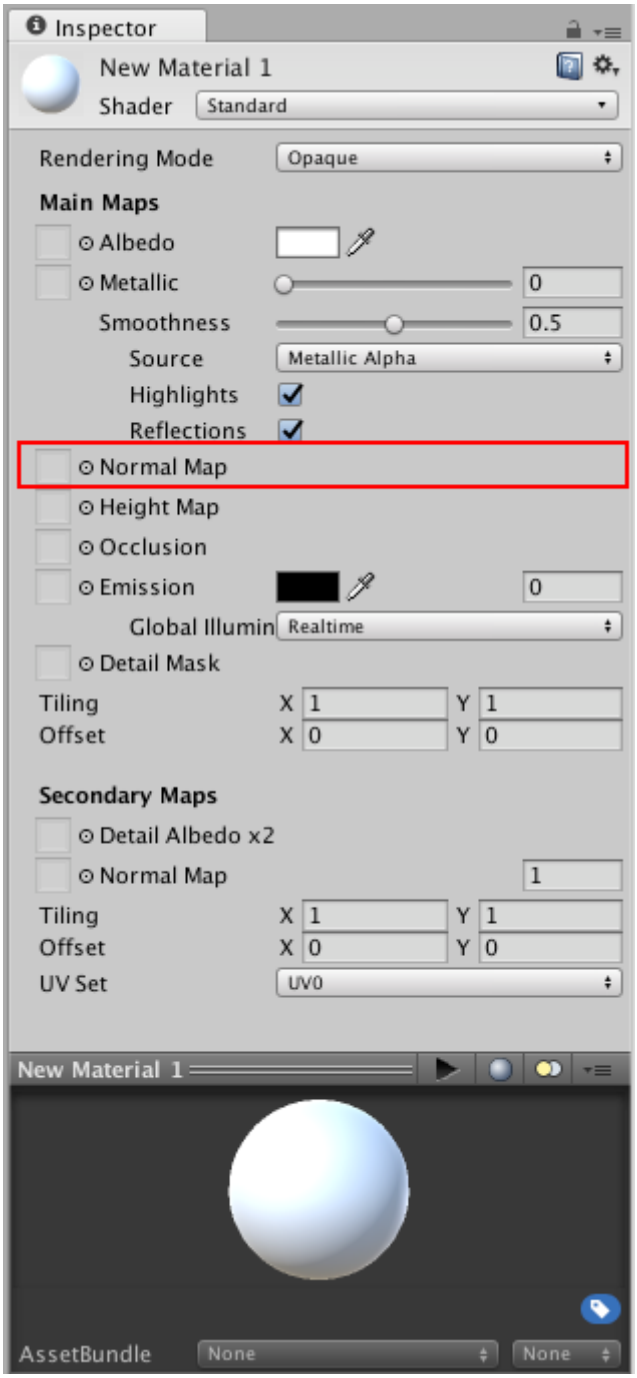
Using a map instead of a slider means you can create materials which include a variety of smoothness levels across the surface (usually designed to match what is shown in the albedo texture).

Property:	*Function:*
Smoothness source	Select the texture channel where the smoothness value is stored.
Specular/Metallic Alpha	Because the smoothness of each point on the surface is a single value, only a single channel of an image texture is required for the data. Therefore the smoothness data is assumed to be stored in the Alpha Channel of the same image texture used for the Metallic or Specular texture map (depending which of these two modes you are using).
Albedo Alpha	This lets you reduce the total number of textures, or use textures of different resolutions for Smoothness and Specular/Metallic.
Highlights	Check this box to disable highlights. This is an optional performance optimization for mobile. It removes the calculation of highlights from the **Standard Shader**. How this affects the appearance mainly depends on the Specular/Metallic value and the Smoothness.
Reflections	Check this box to disable environment reflections. This is an optional performance optimization for mobile. It removes the calculation of highlights from the Standard Shader. Instead of sampling the environment map, an approximation is used. How this affects the appearance depends on the smoothness.

Smoother surfaces are more reflective and have smaller, more tightly-focused specular highlights. Less smooth surfaces do not reflect as much, so specular highlights are less noticable and spread wider across the surface. By matching the specular and smoothness maps to the content in your albedo map, you can begin to create very realistic-looking textures.

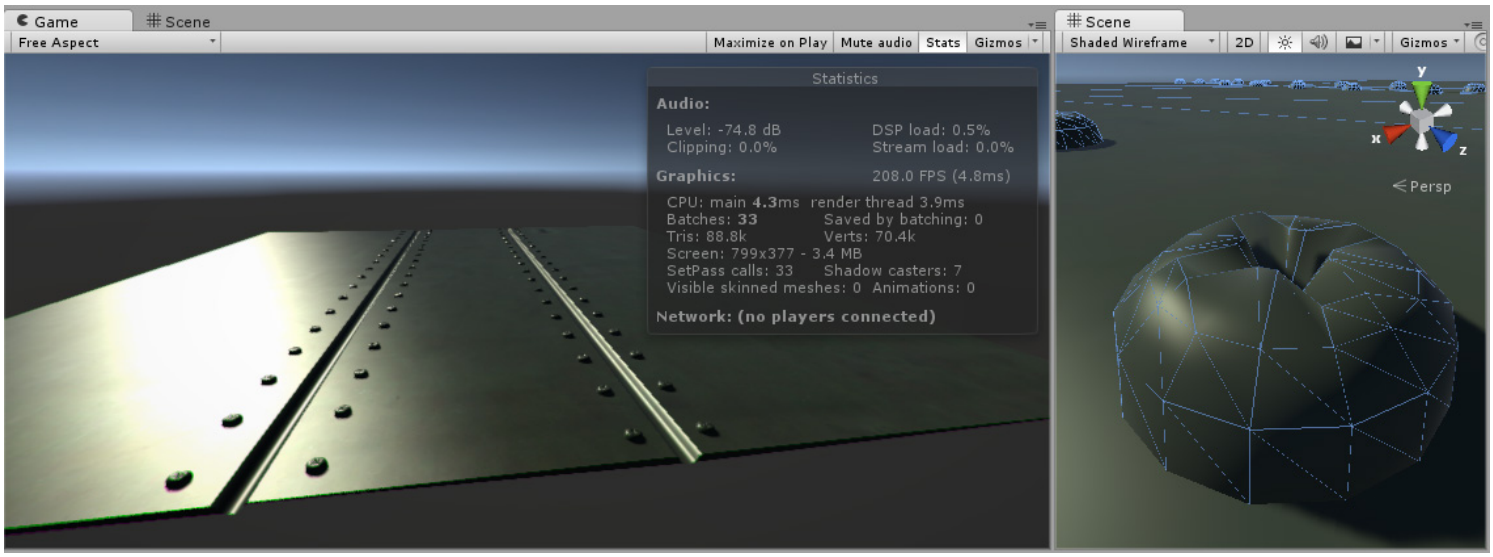
Normal map (Bump mapping)

Leave feedback



Normal maps are a type of **Bump Map** . They are a special kind of texture that allow you to add surface detail such as bumps, grooves, and scratches to a model which catch the light as if they are represented by real geometry.

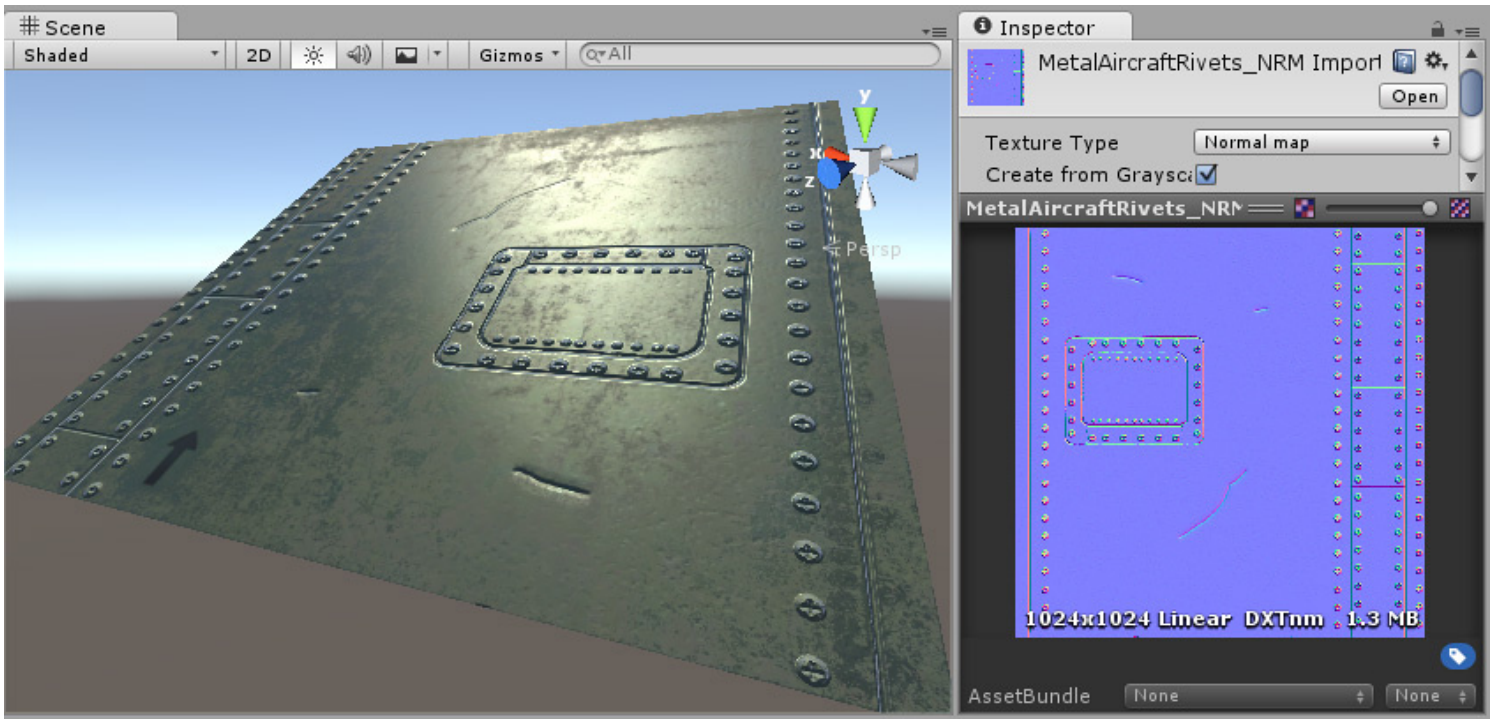
For example, you might want to show a surface which has grooves and screws or rivets across the surface, like an aircraft hull. One way to do this would be to model these details as geometry, as shown below.



A sheet of aircraft metal with details modeled as real geometry.

Depending on the situation it is not normally a good idea to have such tiny details modelled as “real” geometry. On the right you can see the polygons required to make up the detail of a single screwhead. Over a large model with lots of fine surface detail this would require a very high number of polygons to be drawn. To avoid this, we should use a normal map to represent the fine surface detail, and a lower resolution polygonal surface for the larger shape of the model.

If we instead represent this detail with a bump map, the surface geometry can become much simpler, and the detail is represented as a texture which modulates how light reflects off the surface. This is something modern graphics hardware can do extremely fast. Your metal surface can now be a low-poly flat plane, and the screws, rivets, grooves and scratches will catch the light and appear to have depth because of the texture.



The screws, grooves and scratches are defined in a normalmap, which modifies how light reflects off the surface of this low-poly plane, giving the impression of 3D detail. As well as the rivets and screws, a texture allows us to include far more detail like subtle bumps and scratches.

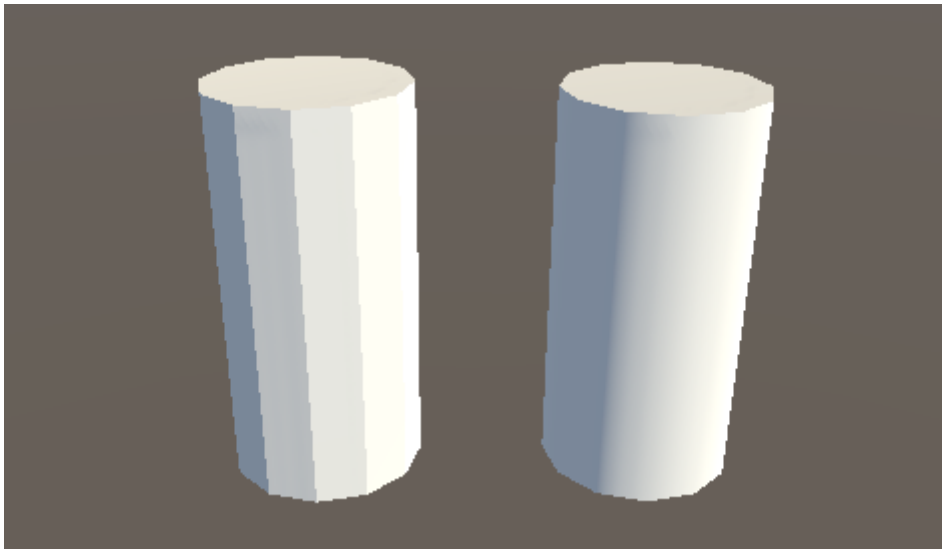
In modern game development art pipelines, artists will use their 3D modelling applications to generate normal maps based on very high resolution source models. The normal maps are then mapped onto a lower-resolution game-ready version of the model, so that the original high-resolution detail is rendered using the normalmap.

How to create and use Bump Maps

Bump mapping is a relatively old graphics technique, but is still one of the core methods required to create detailed realistic realtime graphics. Bump Maps are also commonly referred to as **Normal Maps** or **Height Maps**, however these terms have slightly different meanings which will be explained below.

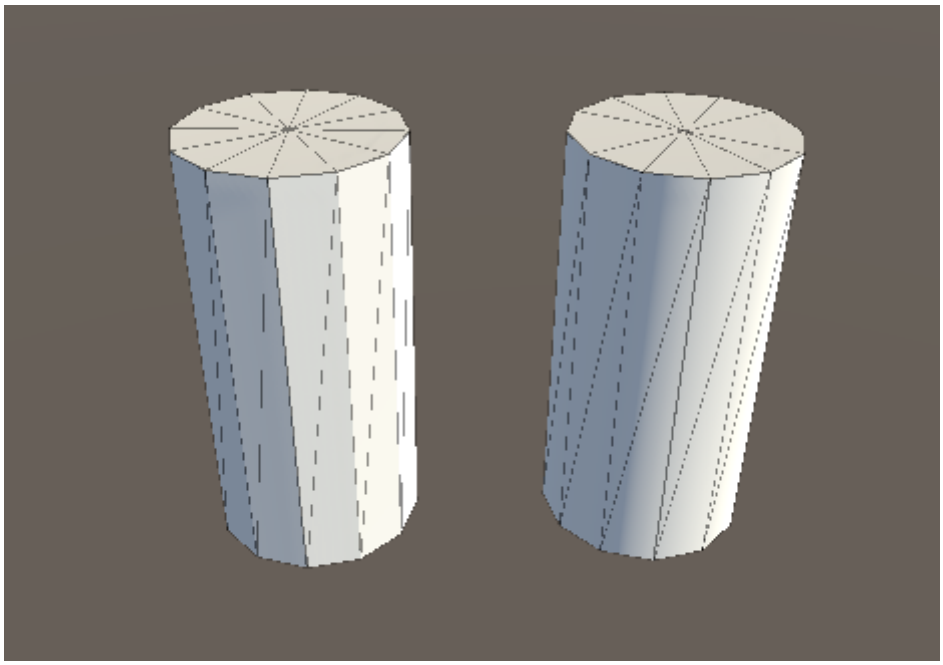
What are Surface Normals?

To really explain how normal mapping works, we will first describe what a "**normal**" is, and how it is used in realtime lighting. Perhaps the most basic example would be a model where each surface polygon is lit simply according to the surface angles relative to the light. The surface angle can be represented as a line protruding in a perpendicular direction from the surface, and this direction (which is a vector) relative to the surface is called a "**surface normal**", or simply, a **normal**.



Two 12-sided cylinders, on the left with flat shading, and on the right with smoothed shading

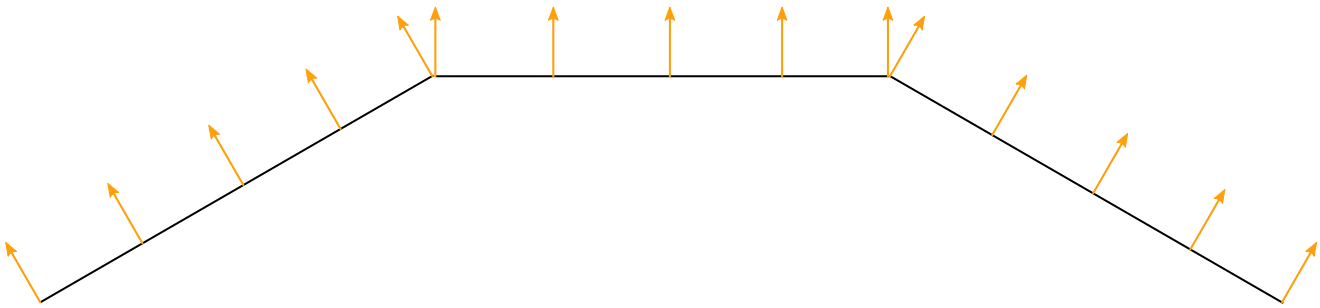
In the image above, the left cylinder has basic flat shading, and each polygon is shaded according to its relative angle to the light source. The lighting on each polygon is constant across the polygon's area because the surface is flat. Here are the same two cylinders, with their wireframe **mesh** visible:



Two 12-sided cylinders, on the left with flat shading, and on the right with smoothed shading

The model on the right has the same number of polygons as the model on the left, however the shading appears smooth - the lighting across the polygons gives the appearance of a curved surface. Why is this? The reason is that the **surface normal** at each point used for reflecting light gradually varies across the width of the polygon, so that for any given point on the surface, the light bounces *as if that surface was curved* and not the flat constant polygon that it really is.

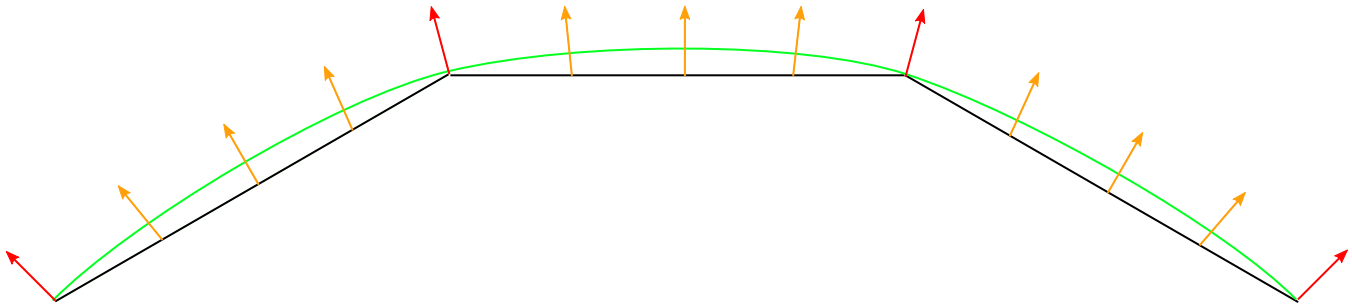
Viewed as a 2D diagram, three of the surface polygons around the outside of the flat-shaded cylinder would look like this:



Flat shading on three polygons, viewed as a 2D diagram

The surface normals are represented by the orange arrows. These are the values used to calculate how light reflects off the surface, so you can see that light will respond the same across the length of each polygon, because the surface normals point in the same direction. This gives the “flat shading”, and is the reason the left cylinder’s polygons appear to have hard edges.

For the smooth shaded cylinder however, the surface normals vary across the flat polygons, as represented here:



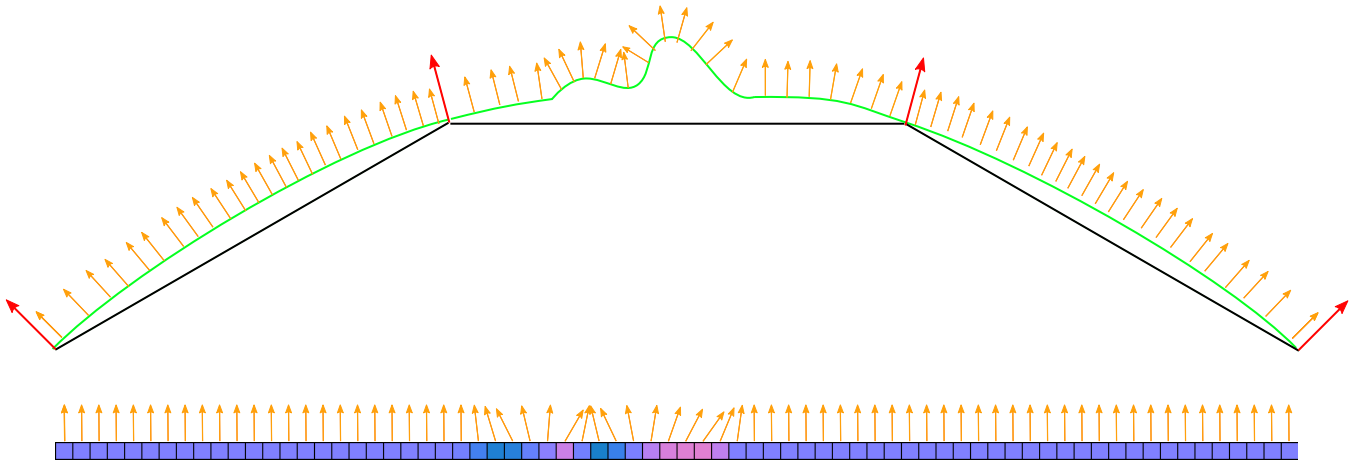
Smooth shading on three polygons, viewed as a 2D diagram

The normal directions gradually change across the flat polygon surface, so that the shading across the surface gives the impression of a smooth curve (as represented by the greeen line). This does not affect the actual polygonal nature of the mesh, only how the lighting is calculated on the flat surfaces. This apparent curved surface is not really present, and viewing the faces at glancing angles will reveal the true nature of the flat polygons, however from most viewing angles the cylinder appears to have a smooth curved surface.

Using this basic smooth shading, the data determining the normal direction is actually only stored **per vertex**, so the changing values across the surface are interpolated from one vertex to the next. In the diagram above, the red arrows indicate the stored normal direction at each vertex, and the orange arrows indicate examples of the interpolated normal directions across the area of the polygon.

What is Normal mapping?

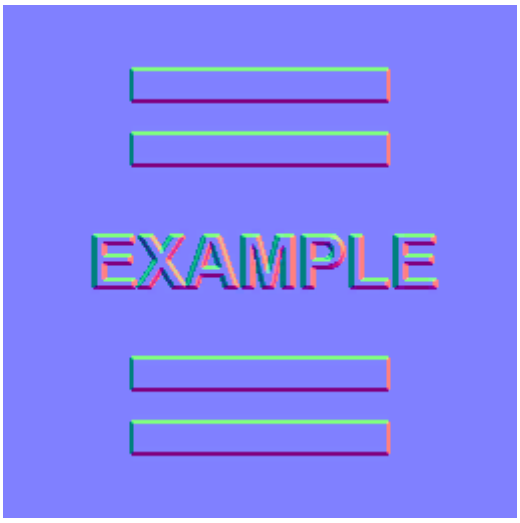
Normal mapping takes this modification of surface normals one step further, by using a texture to store information about how to modify the surface normals across the model. A normal map is an image texture mapped to the surface of a model, similar to regular colour textures, however each **pixel** in the texture of the normal map (called a **texel**) represents a deviation in surface normal direction away from the “true” surface normal of the flat (or smooth interpolated) polygon.



Normal mapping across three polygons, viewed as a 2D diagram

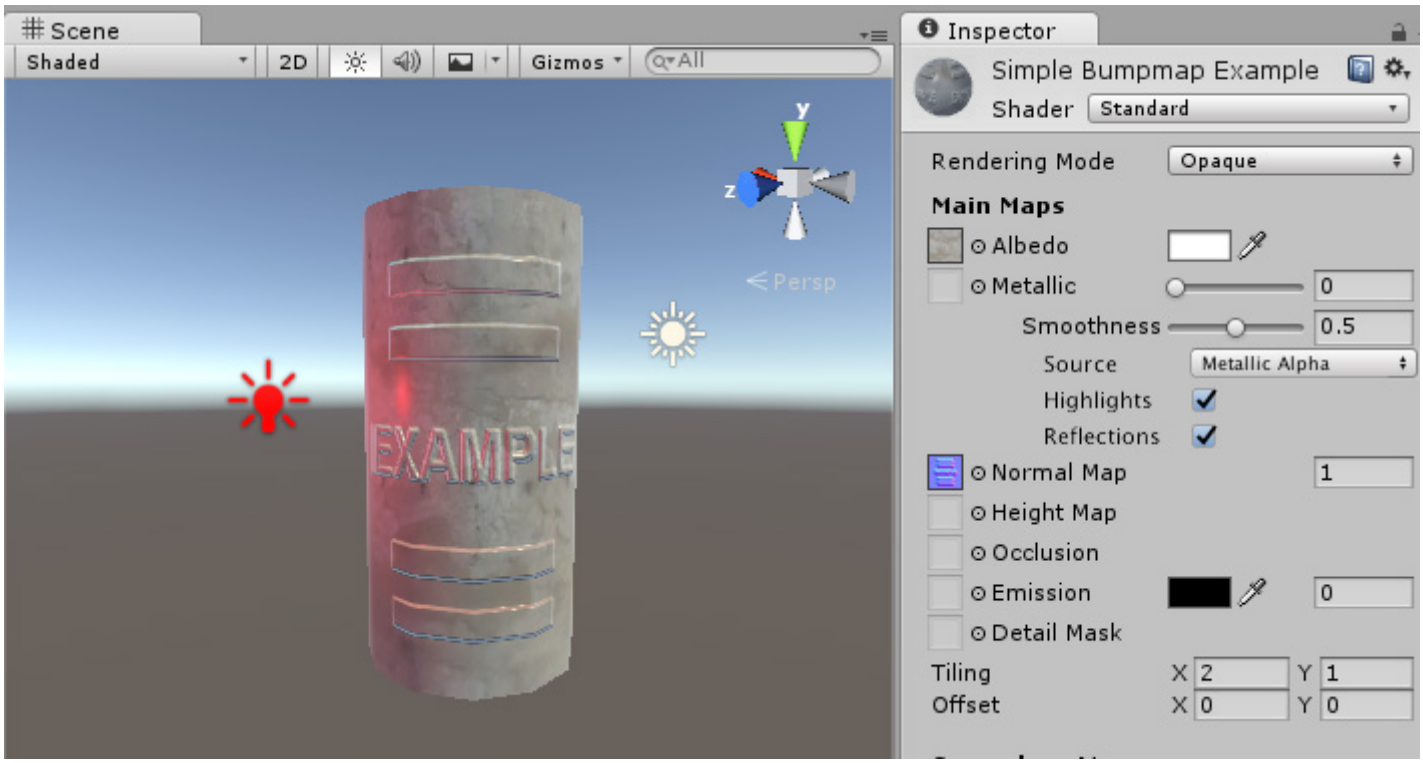
In this diagram, which is again a 2D representation of three polygons on the surface of a 3D model, each orange arrow corresponds to a pixel in the normalmap texture. Below, is a single-pixel slice of a normalmap texture. In the centre, you can see the normals have been modified, giving the *appearance* of a couple of bumps on the surface of the polygon. These bumps would only be apparent due to the way lighting appears on the surface, because these modified normals are used in the lighting calculations.

The colours visible in a raw normal map file typically have a blueish hue, and don't contain any actual light or dark shading - this is because the colours themselves are not intended to be displayed as they are. Instead, the RGB values of each texel represent the X,Y & Z values of a direction vector, and are applied as a modification to the basic interpolated smooth normals of the polygon surfaces.



An example normal map texture

This is a simple normal map, containing the bump information for some raised rectangles and text. This normal map can be imported into Unity and placed into Normal Map slot of the **Standard Shader**. When combined in a material with a colour map (the Albedo map) and applied to the surface of of the cylinder mesh above, the result looks like this:



The example normal map applied to the surface of the cylinder mesh used above

Again, this does not affect the actual polygonal nature of the mesh, only how the lighting is calculated on the surfaces. This apparent raised lettering and shapes on the surface are not really present, and viewing the faces at glancing angles will reveal the true nature of the flat surface, however from most viewing angles the cylinder now appears to have embossed detail raised off the surface.

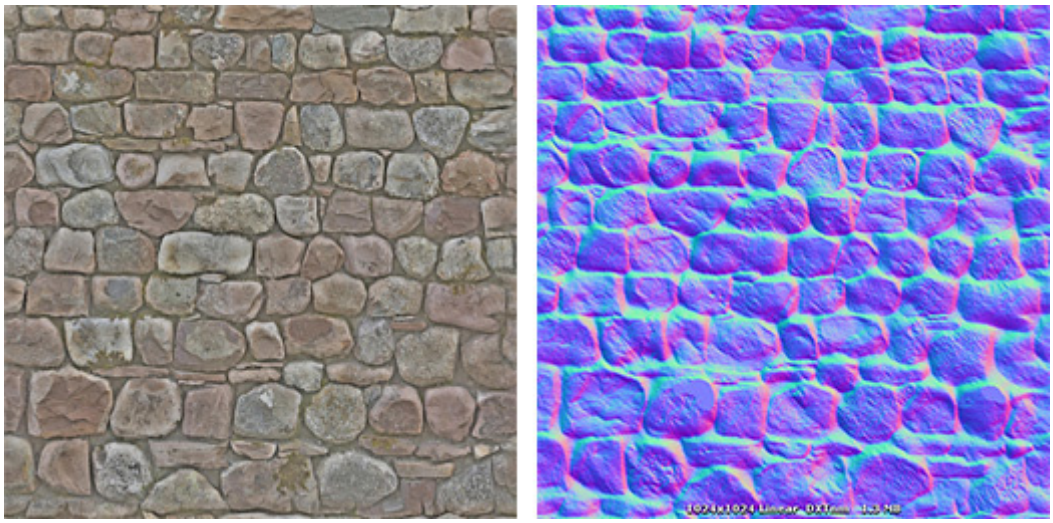
How do I get or make normal maps?

Commonly, Normal Maps are produced by 3D or Texture artists in conjunction with the model or textures they are producing, and they often mirror the layout and contents of the Albedo map. Sometimes they are produced by hand, and sometimes they are rendered out from a 3D application.

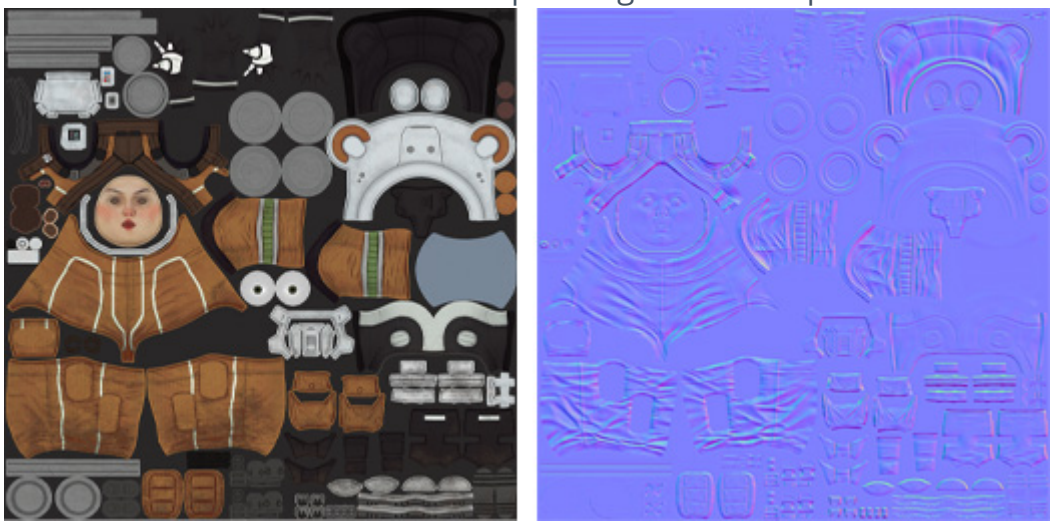
How to render normal maps from a 3D application is beyond the scope of this documentation, however the basic concept is that a 3D artist would produce two versions of a model - a very high resolution model containing all detail as polygons, and a lower-res “game ready” model. The high res model would be too detailed to run optimally in a game (too many triangles in the mesh), but it is used in the 3D modelling application to generate the normal maps. The lower-res version of model can then omit the very fine level of geometry detail that is now stored in the normal maps, so that it can be rendered using the normal mapping instead. A typical use case for this would be to show the bumped detail of creases, buttons, buckles and seams on a characters clothing.

There are some software packages which can analyse the lighting in a regular photographic texture, and extract a normalmap from it. This works by assuming the original texture is lit from a constant direction, and the light and dark areas are analysed and assumed to correspond with angled surfaces. However, when actually using a bump map, you need to make sure that your Albedo texture *does not* have lighting from any particular direction in the image - ideally it should represent the colours of the surface with no lighting at all - because the lighting information will be calculated by Unity according to the light direction, surface angle and bump map information.

Here are two examples, one is a simple repeating stone wall texture with its corresponding normal map, and one is a character’s texture atlas with its corresponding normal map:



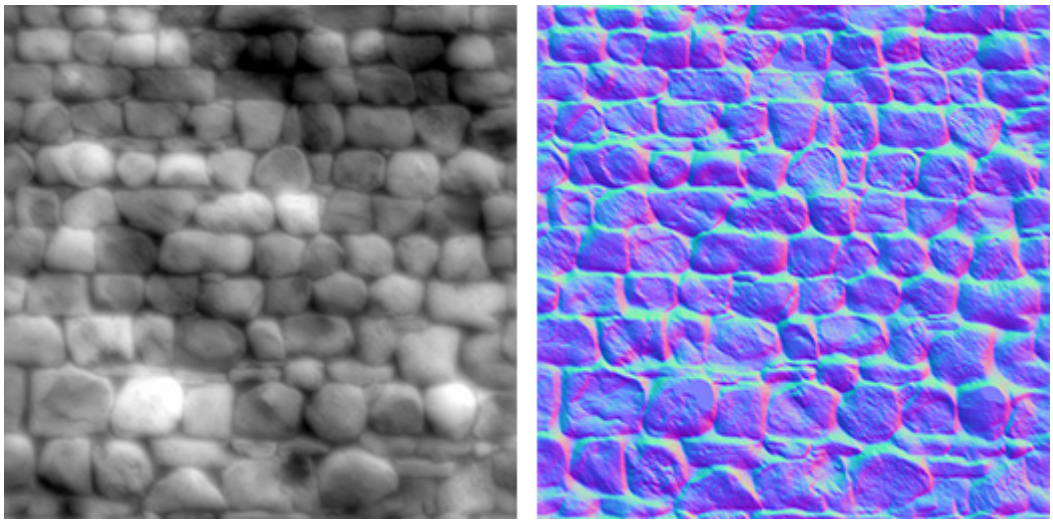
A stone wall texture and its corresponding normal map texture



A character texture atlas, and its corresponding normal map texture atlas

What’s the difference between Bump Maps, Normal Maps and Height Maps?

Normal Maps and **Height Maps** are both *types* of Bump Map. They both contain data for representing apparent detail on the surface of simpler polygonal meshes, but they each store that data in a different way.



On the left, a height map for bump mapping a stone wall. On the right, a normal map for bump mapping a stone wall.

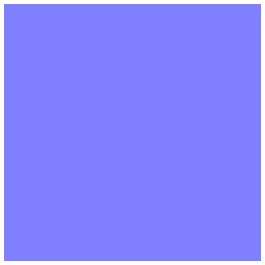
Above, on the left, you can see a height map used for bump mapping a stone wall. A height map is a simple black and white texture, where each pixel represents the amount that point on the surface should appear to be raised. The whiter the pixel colour, the higher the area appears to be raised.

A normal map is an RGB texture, where each pixel represents the *difference in direction* the surface should appear to be facing, relative to its un-modified surface normal. These textures tend to have a bluey-purple tinge, because of the way the vector is stored in the RGB values.

Modern realtime 3D graphics hardware rely on Normal Maps, because they contain the vectors required to modify how light should appear to bounce of the surface. Unity can also accept Height Maps for bump mapping, but they must be converted to Normal Maps on import in order to use them.

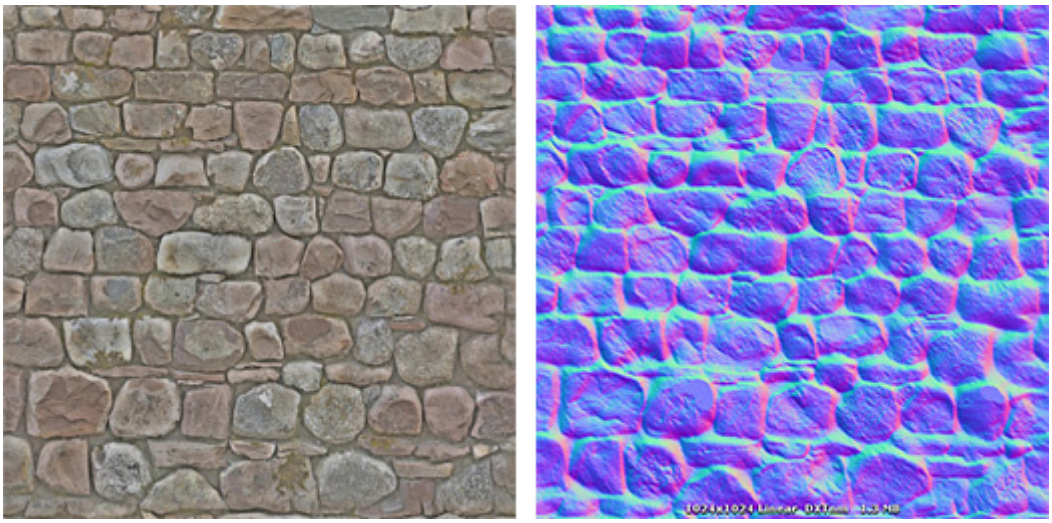
Why the bluey-purple colours?

Understanding this is not vital for using normal maps! It's ok to skip this paragraph. However if you really want to know: The RGB colour values are used to store the X,Y,Z direction of the vector, with Z being "up" (contrary to Unity's usual convention of using Y as "up"). In addition, the values in the texture are treated as having been halved, with 0.5 added. This allows vectors of all directions to be stored. Therefore to convert an RGB colour to a vector direction, you must multiply by two, then subtract 1. For example, an RGB value of (0.5, 0.5, 1) or #8080FF in hex results in a vector of (0,0,1) which is "up" for the purposes of normal-mapping - and represents no change to the surface of the model. This is the colour you see in the flat areas of the "example" normal map earlier on this page.



A normal map using only #8080FF, which translates to a normal vector of 0,0,1 or "straight up". This applies no modification to the surface normal of the polygon, and therefore produces no change to the lighting. Any pixels which are different to this colour results in a vectors that point in a different direction - which therefore modify the angle that is used to calculate light bounce at that point.

A value of (0.43, 0.91, 0.80) gives a vector of (–0.14, 0.82, 0.6), which is quite a steep modification to the surface. Colours like this can be seen in the bright cyan areas of the stone wall normal map at the top of some of the stone edges. The result is that these edges catch the light at a very different angle to the flatter faces of the stones.

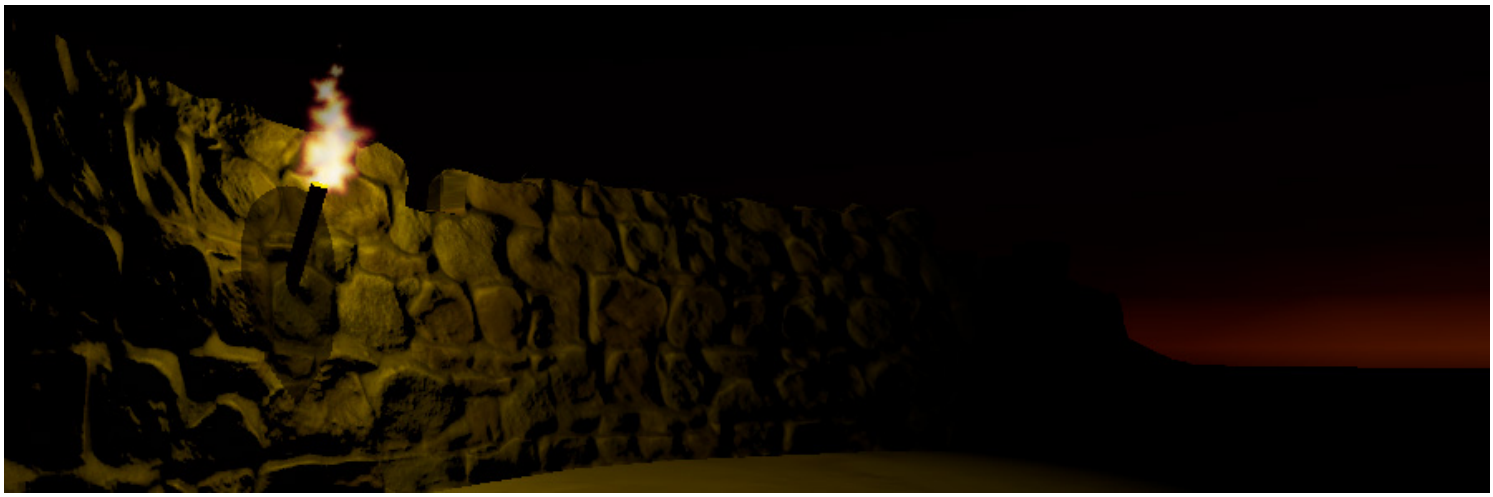


The bright cyan areas in the normalmap for these stones show a steep modification to the polygon's surface normals at the top edge of each stone, causing them to catch the light at the correct angle.

Normal maps

A stone wall with no bumpmap effect. The edges and facets of the rock do not catch the directional sun light in the scene.

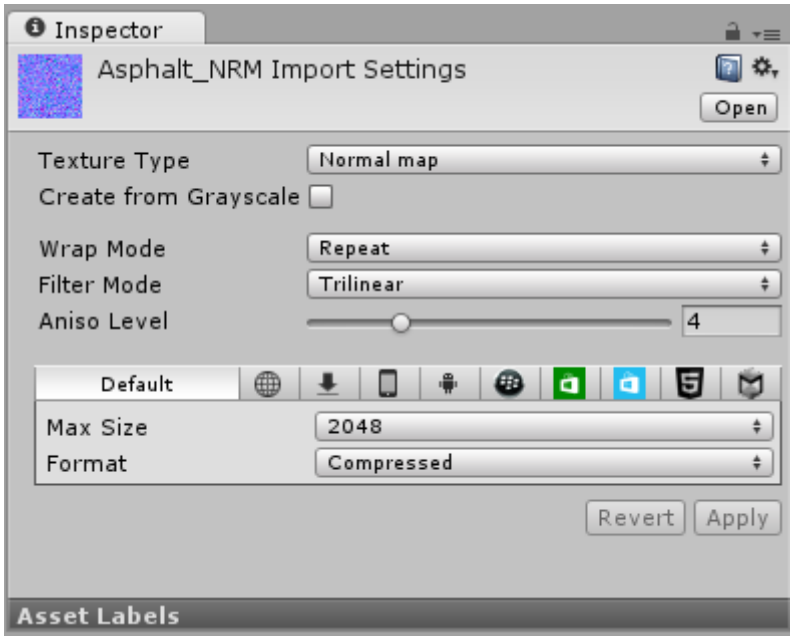
The same stone wall with bumpmapping applied. The edges of the stones facing the sun reflect the directional sun light very differently to the faces of the stones, and the edges facing away.



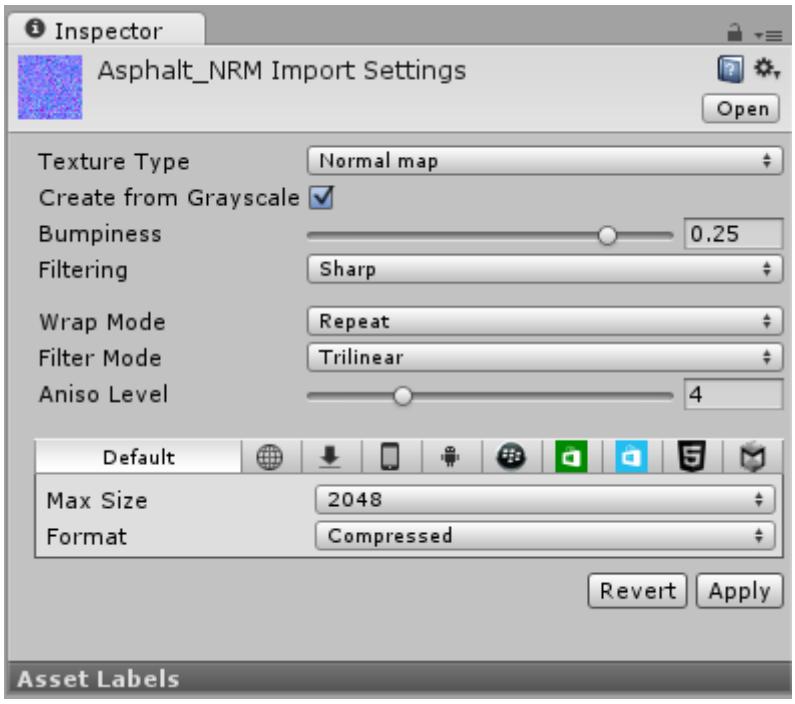
The same bumpmapped stone wall, in a different lighting scenario. A point light torch illuminates the stones. Each pixel of the stone wall is lit according to how the light hits the angle of the base model (the polygon), adjusted by the vectors in the normal maps. Therefore pixels facing the light are bright, and pixels facing away from the light are darker, or in shadow.

How to import and use Normal Maps and Height Maps

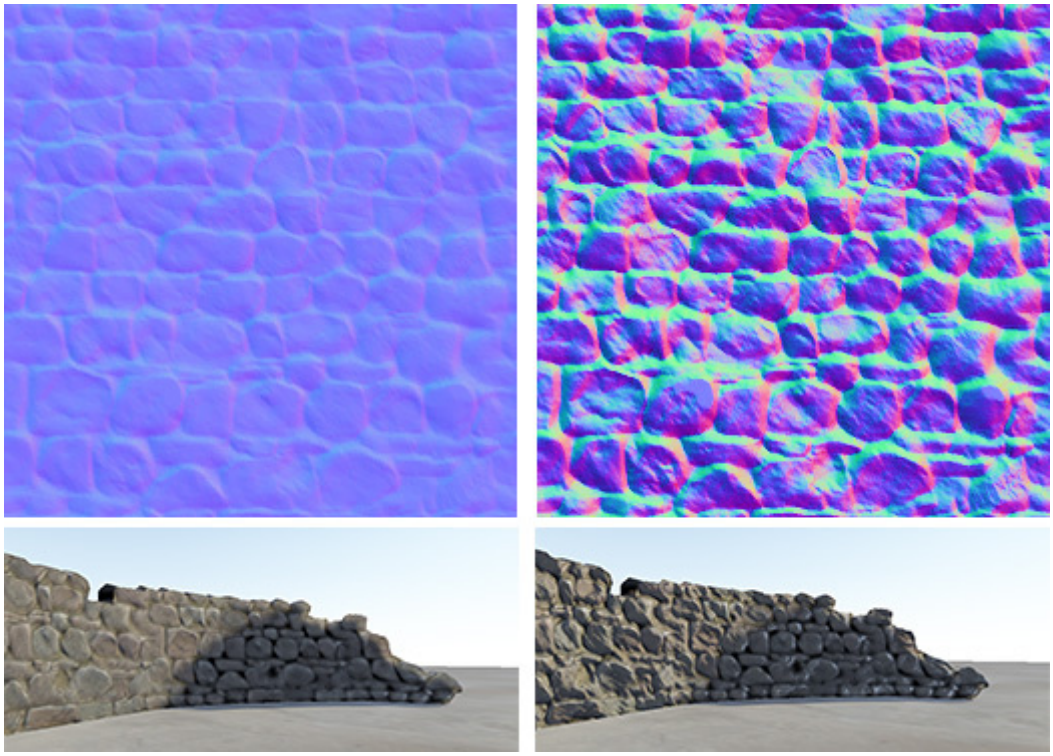
A normal map can be imported by placing the texture file in your **assets** folder, as usual. However, you need to tell Unity that this texture is a normal map. You can do this by changing the “Texture Type” setting to “Normal Map” in the import **inspector** settings.



To import a black and white heightmap as a normal map, the process is almost identical, except you need to check the “Create from Greyscale” option.

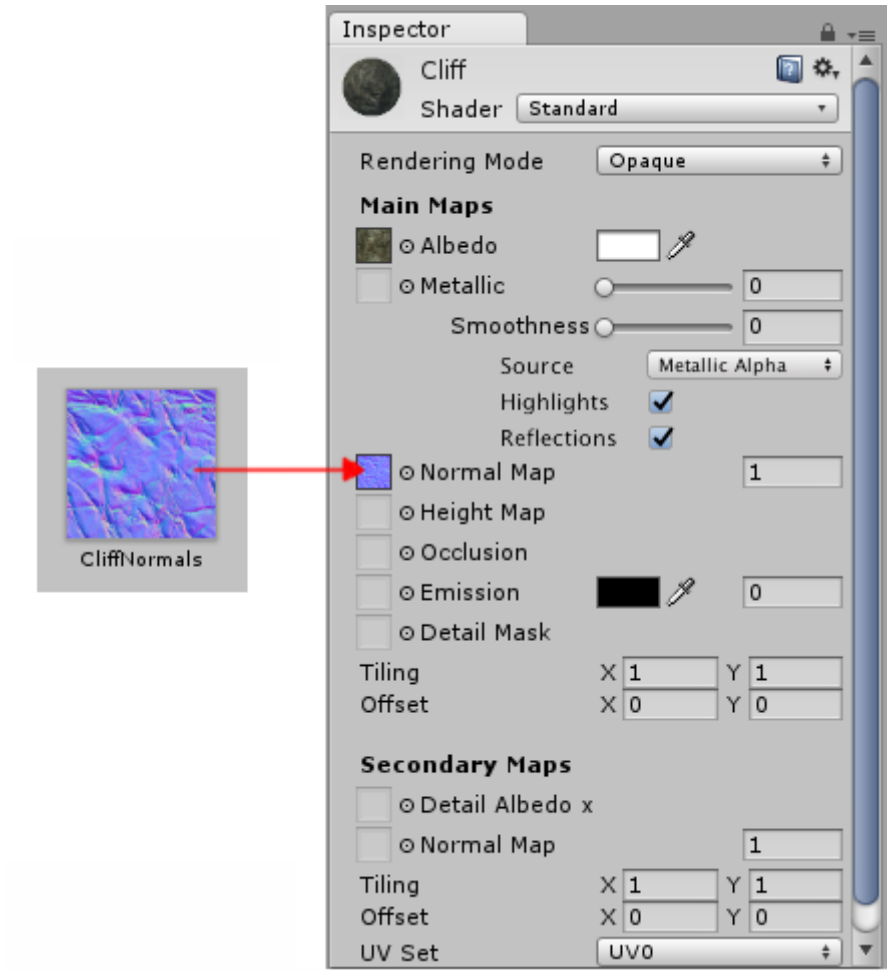


With “Create From Greyscale” selected, a Bumpiness slider will appear in the inspector. You can use this to control how steep the angles in the normalmap are, when being converted from the heights in your heightmap. A low bumpiness value will mean that even sharp contrast in the heightmap will be translated to gentle angles and bumps. A high value will create exaggerated bumps and very high contrast lighting responses to the bumps.



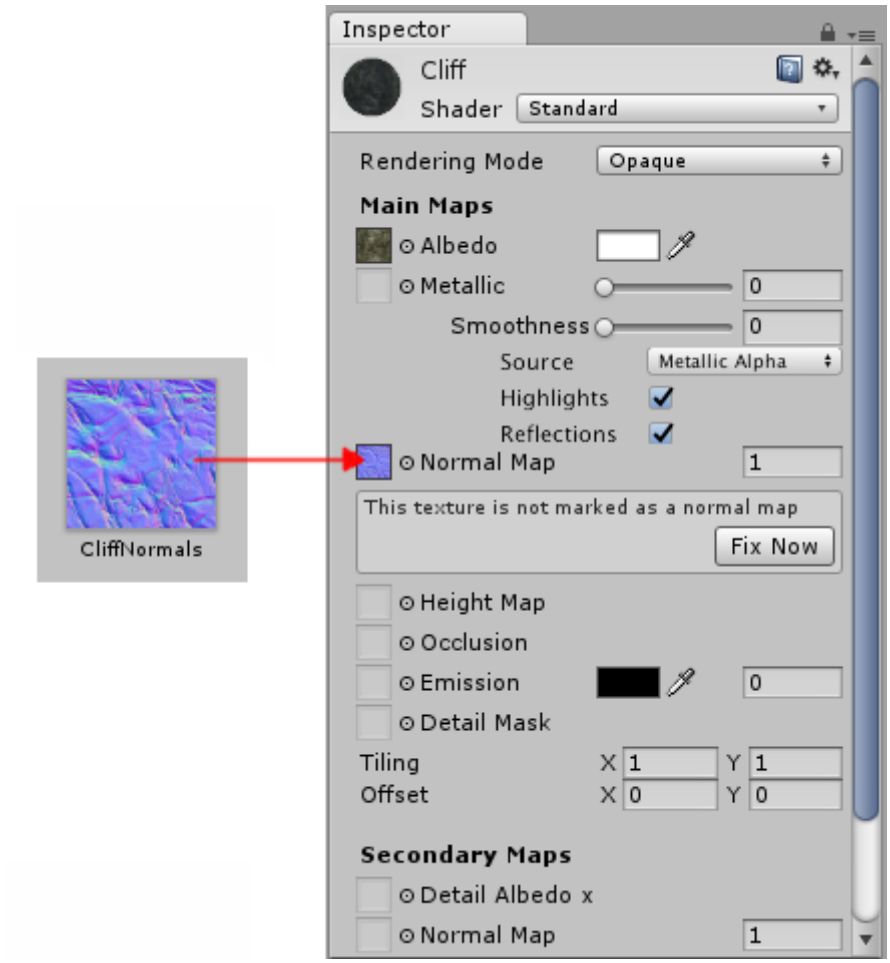
Low and High Bumpiness settings when importing a height map as a normal map, and the resulting effect on the model.

Once you have a normalmap in your assets, you can place it into the Normal Map slot of your Material in the inspector. The Standard **Shader** has a normal map slot, and many of the older legacy shaders also support normal maps.



Placing a normal map texture into the correct slot in a material using the Standard Shader

If you imported a normalmap or heightmap, and did not mark it as a normal map (By selecting **Texture Type: Normal Map** as described above), the Material inspector will warn you about this and offer to fix it, as so:



The “Fix Now” warning appears when trying to use a normalmap that has not been marked as such in the inspector.

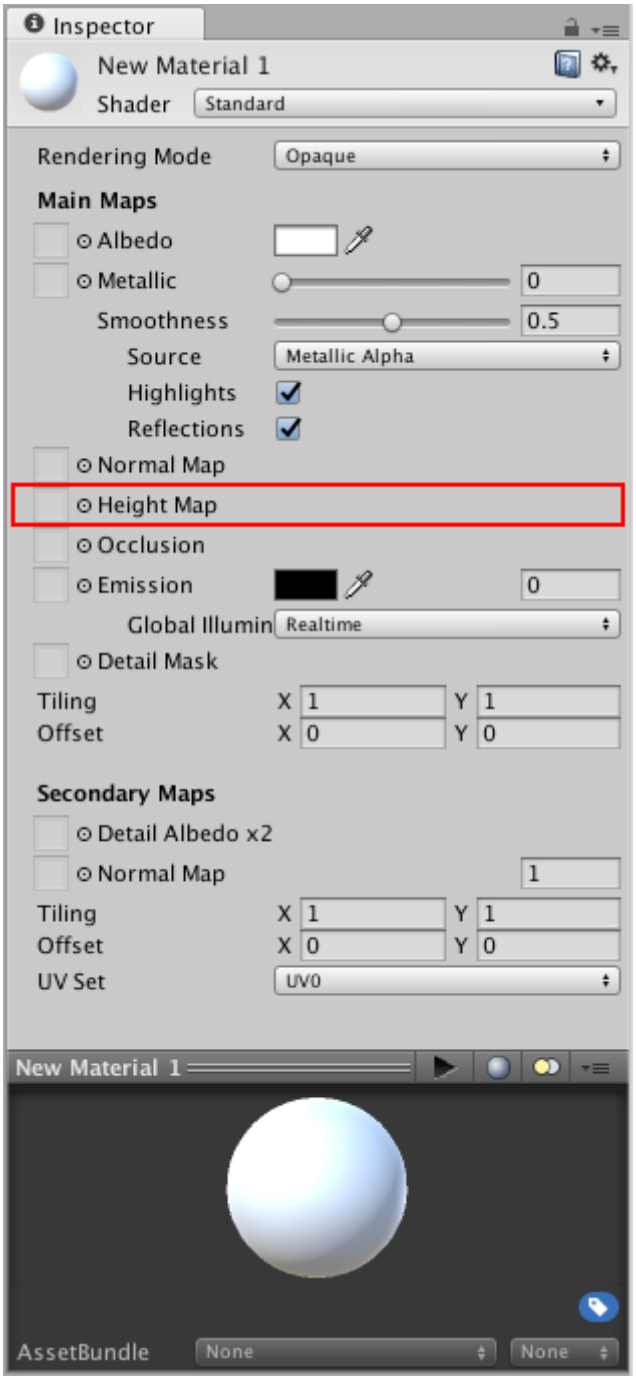
Clicking “Fix Now” has the same effect as selecting **Texture Type: Normal Map** in the texture inspector settings. This will work if your texture really is a normal map. However if it is a greyscale heightmap, it will not automatically detect this - so for heightmaps you must always select the “Create from Greyscale” option in the texture’s inspector window.

Secondary Normal Maps

You may also notice that there is a second Normal Map slot further down in the Material inspector for the Standard Shader. This allows you to use an additional normal map for creating extra detail. You can add a normal map into this slot in the same way as the regular normal map slot, but the intention here is that you should use a different scale or frequency of tiling so that the two normal maps together produce a high level of detail at different scales. For example, your regular normal map could define the details of panelling on a wall or vehicle, with groves for the panel edges. A secondary normal map could provide very fine bump detail for scratches and wear on the surface which may be tiled at 5 to 10 times the scale of the base normal map. These details could be so fine as to only be visible when examined closely. To have this amount of detail on the base normal map would require the base normal map to be incredibly large, however by combining two at different scales, a high overall level of detail can be achieved with two relatively small normal map textures.

Heightmap

Leave feedback

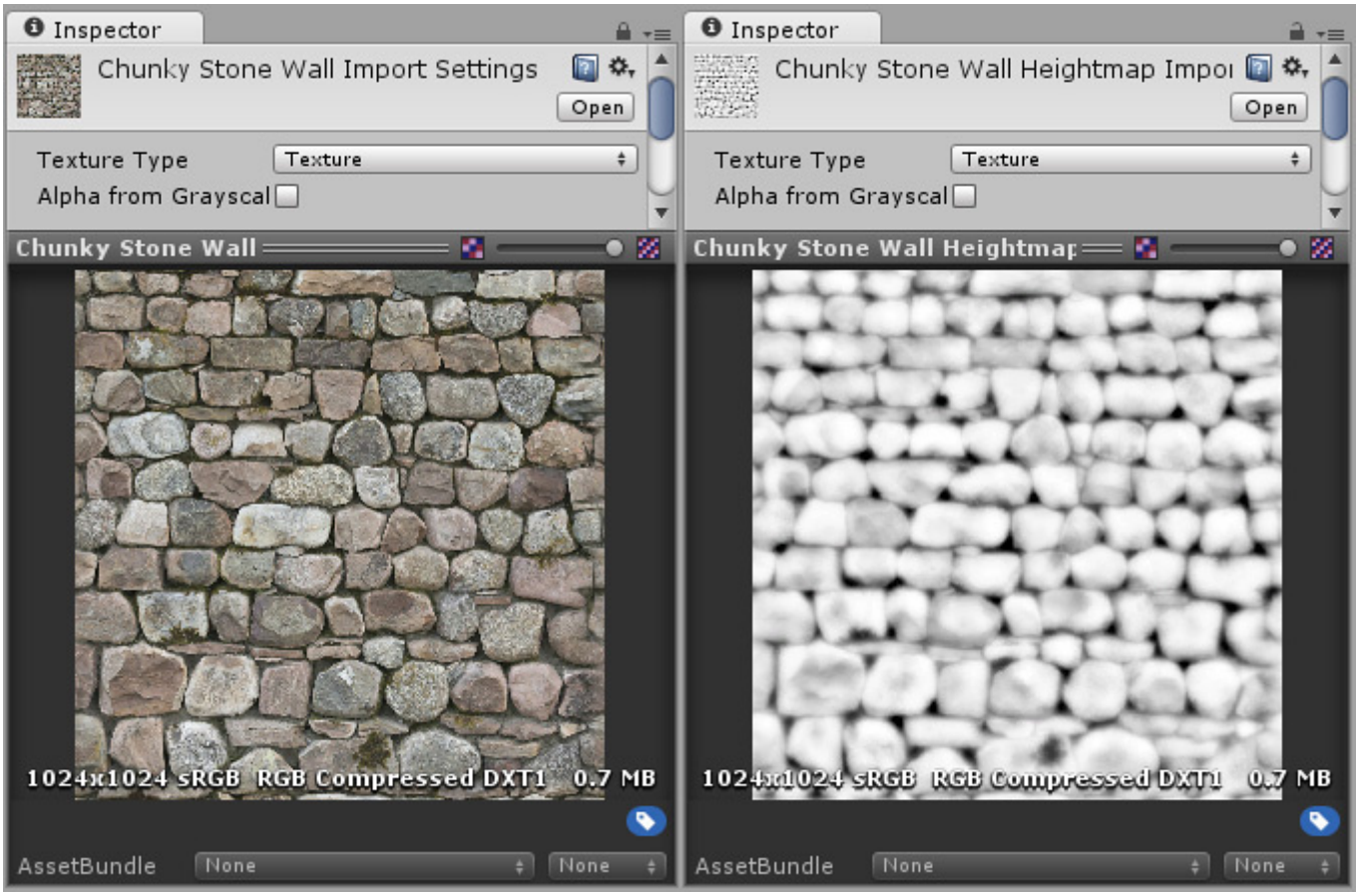


Height mapping (also known as parallax mapping) is a similar concept to normal mapping, however this technique is more complex - and therefore also more performance-expensive. Heightmaps are usually used in conjunction with normalmaps, and often they are used to give extra definition to surfaces where the texture maps are responsible for **rendering** large bumps and protrusions.

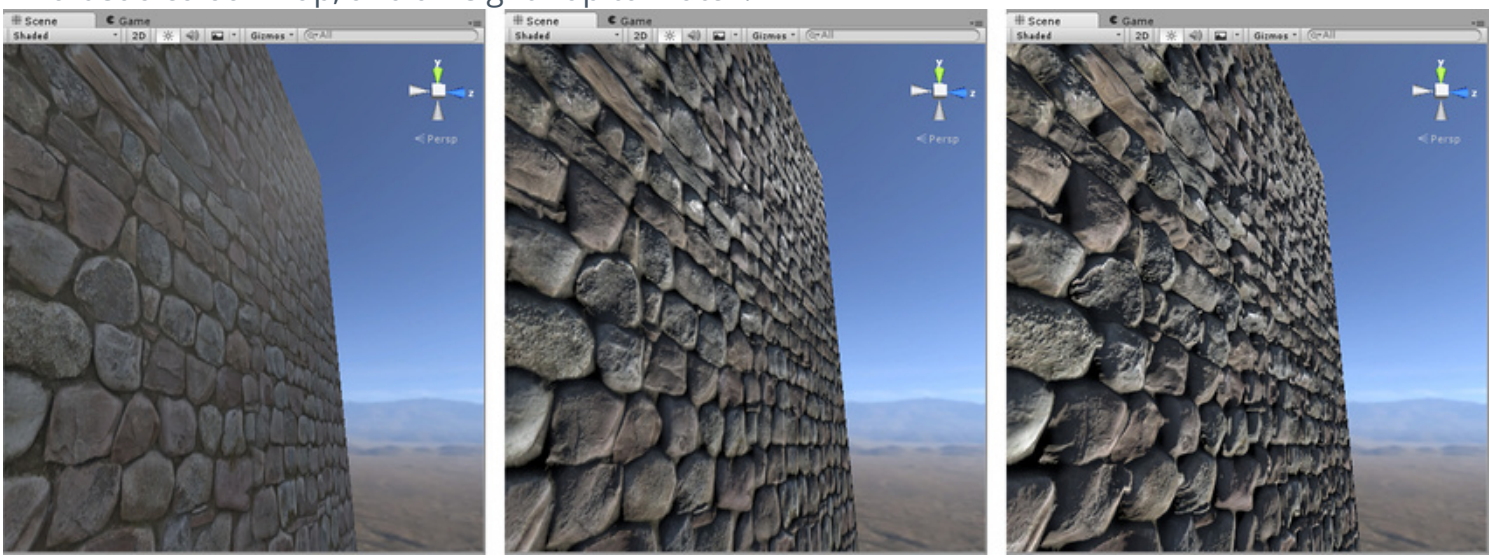
While normal mapping modifies the lighting across the surface of the texture, parallax height mapping goes a step further and actually shifts the areas of the visible surface texture around, to achieve a kind of surface-level occlusion effect. This means that apparent bumps will have their near side (facing the camera) expanded and exaggerated, and their far side (facing away from the camera) will be reduced and seem to be occluded from view.

This effect, while it can produce a very convincing representation of 3D geometry, is still limited to the surface of the flat polygons of an object's **mesh**. This means that while surface bumps will appear to protrude and occlude each other, the "silhouette" of the model will never be modified, because ultimately the effect is drawn onto the surface of the model and does not modify the actual geometry.

A heightmap should be a greyscale image, with white areas representing the high areas of your texture and black representing the low areas. Here's a typical albedo map and a heightmap to match.



An albedo colour map, and a heightmap to match.



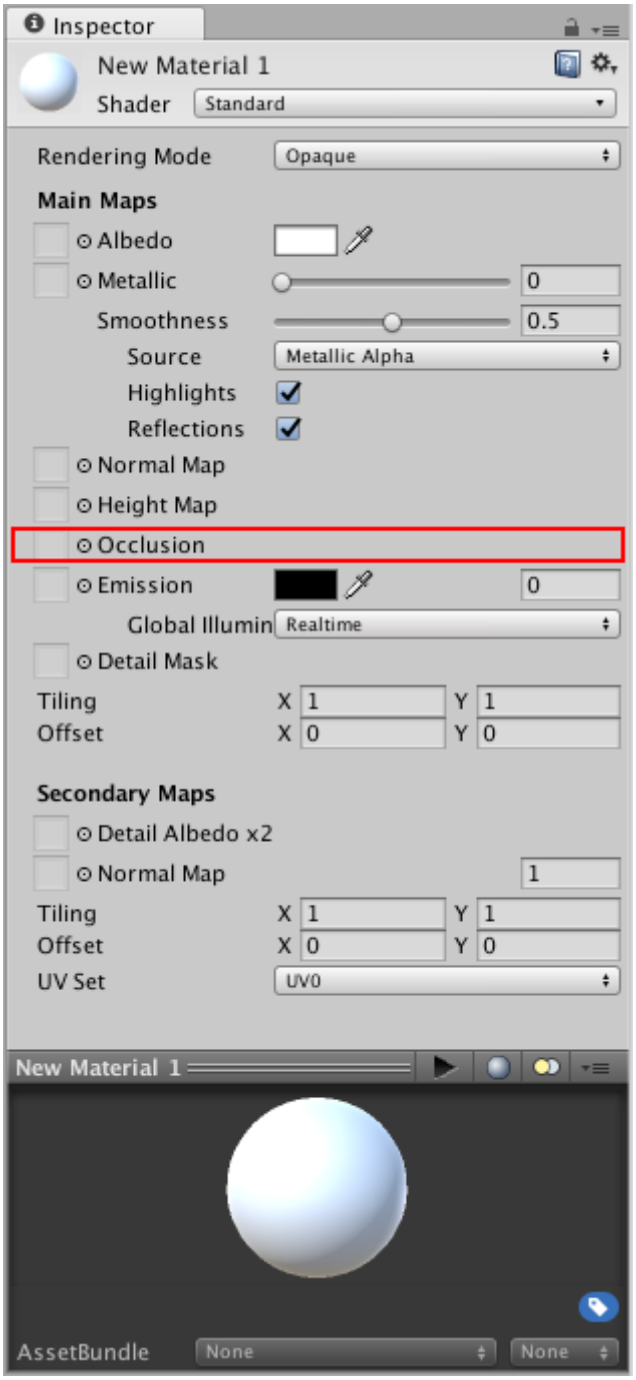
From left to right in the above image: 1. A rocky wall material with albedo assigned, but no normalmap or heightmap. 2. The normal assigned. Lighting is modified on the surface, but rocks do not occlude each other. 3.

The final effect with normalmap and heightmap assigned. The rocks appear to protrude out from the surface, and nearer rocks seem to occlude rocks behind them.

Often (but not always) the greyscale image used for a heightmap is also a good image to use for the occlusion map. For information on occlusion maps, see the next section.

Occlusion Map

Leave feedback



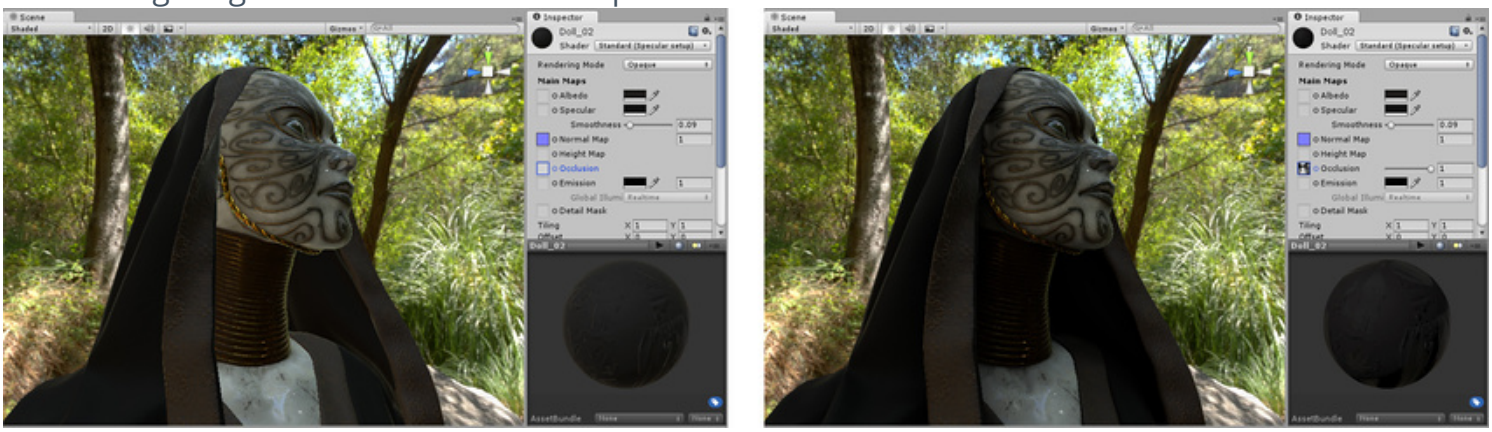
The occlusion map is used to provide information about which areas of the model should receive high or low indirect lighting. Indirect lighting comes from ambient lighting and reflections, and so steep concave parts of your model such as a crack or fold would not realistically receive much indirect light.

Occlusion texture maps are normally calculated by 3D applications directly from the 3D model using the modeller or third party software.

An occlusion map is a greyscale image, with white indicating areas that should receive full indirect lighting, and black indicating no indirect lighting. Sometimes this is as simple as a greyscale heightmap, for simple surfaces (such as the knobbly stone wall texture shown in the heightmap example above).

At other times, generating the correct occlusion texture is slightly more complex. For example, if a character in your **scene** is wearing a hood, the inside edges of the hood should be set to very low indirect lighting, or none at all. In these situations, occlusion maps will be often be produced by artists, using 3D applications to automatically generate an occlusion map based on the model.

This occlusion map identifies areas on a character's sleeve that are exposed or hidden from ambient lighting. It is used on the model pictured below.

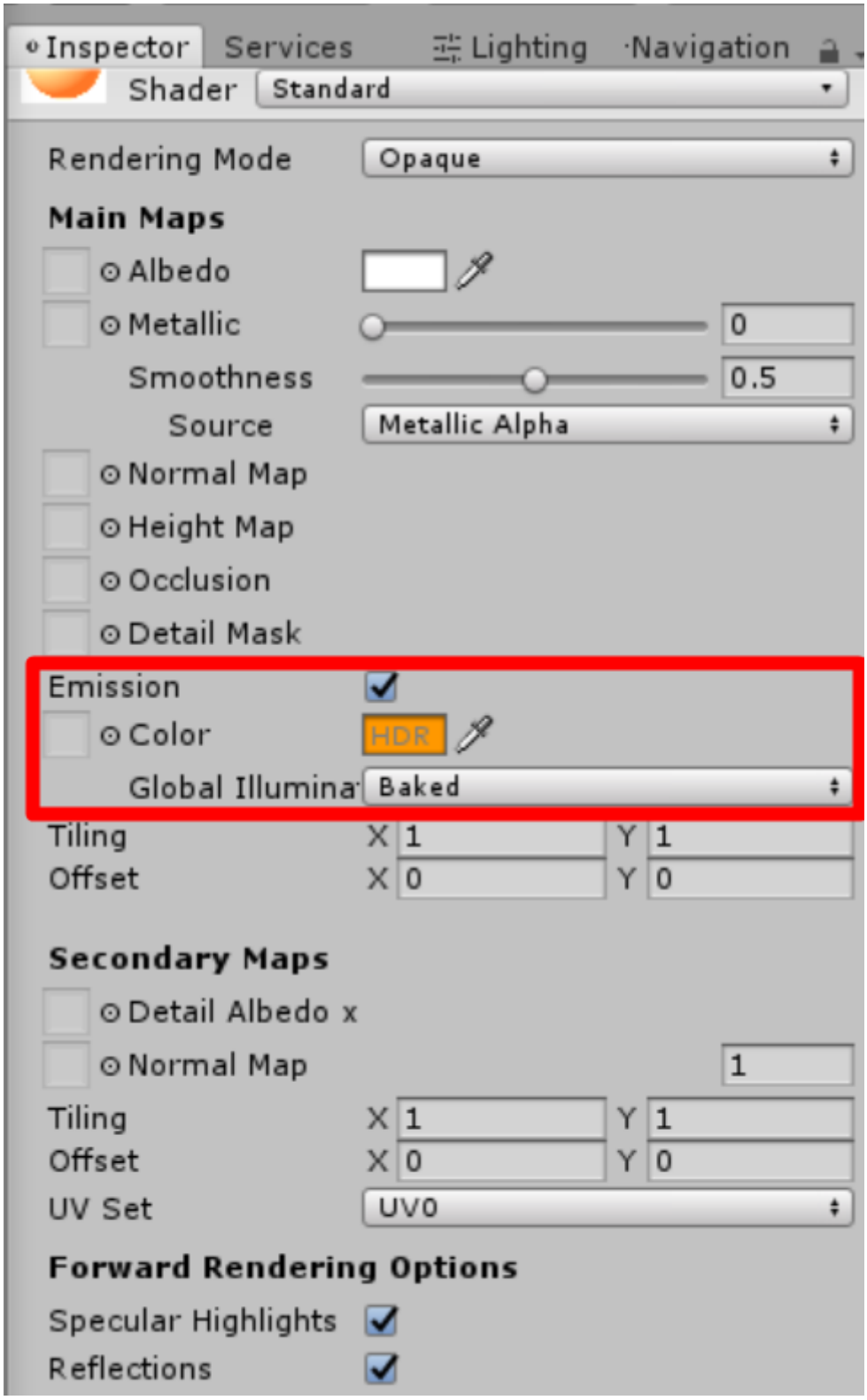


Before and after applying an occlusion map. The areas that are partially obscured, particularly in the folds of fabric around the neck, are lit too brightly on the left. After the ambient occlusion map

is assigned, these areas are no longer lit by the green ambient light from the surrounding wooded environment.

Emission

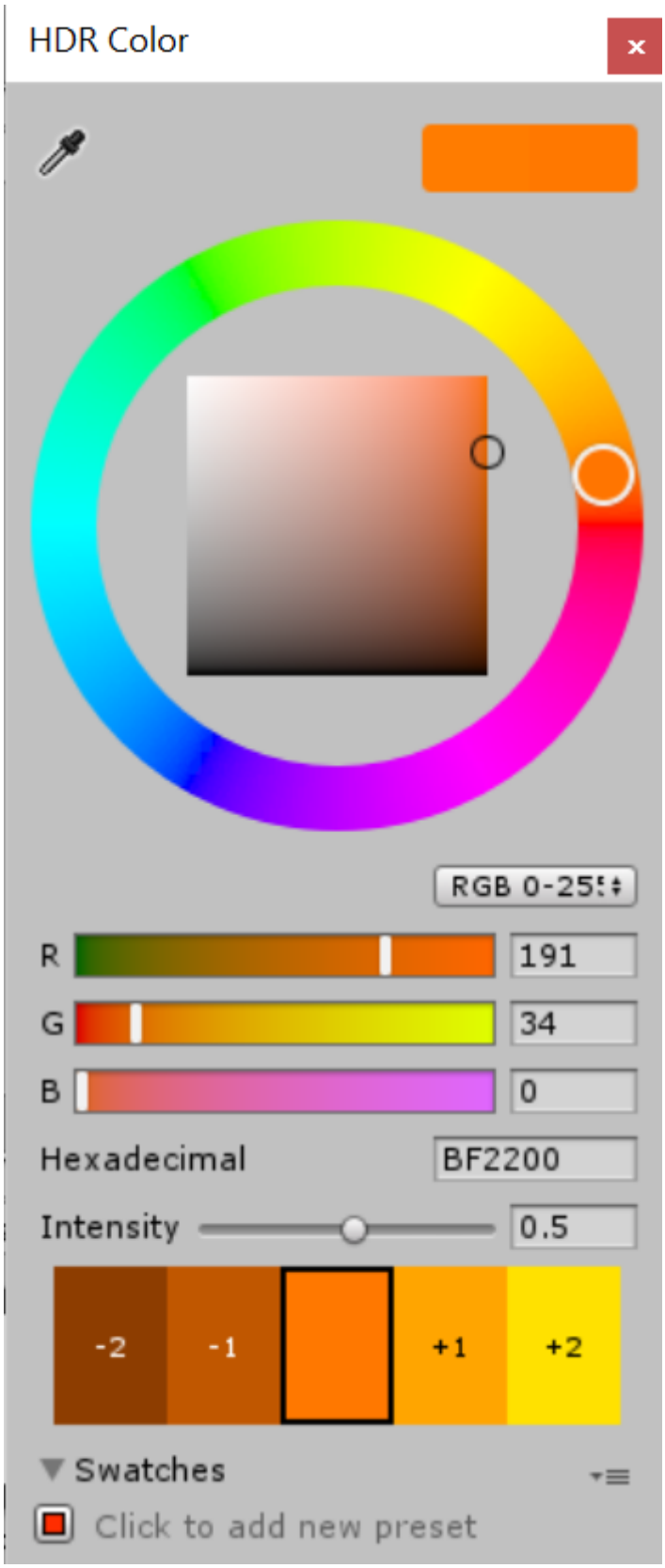
Leave feedback



Emission controls the color and intensity of light emitted from the surface. When you use an emissive Material in your **Scene**, it appears as a visible source of light. The **GameObject** appears to be self-illuminated.

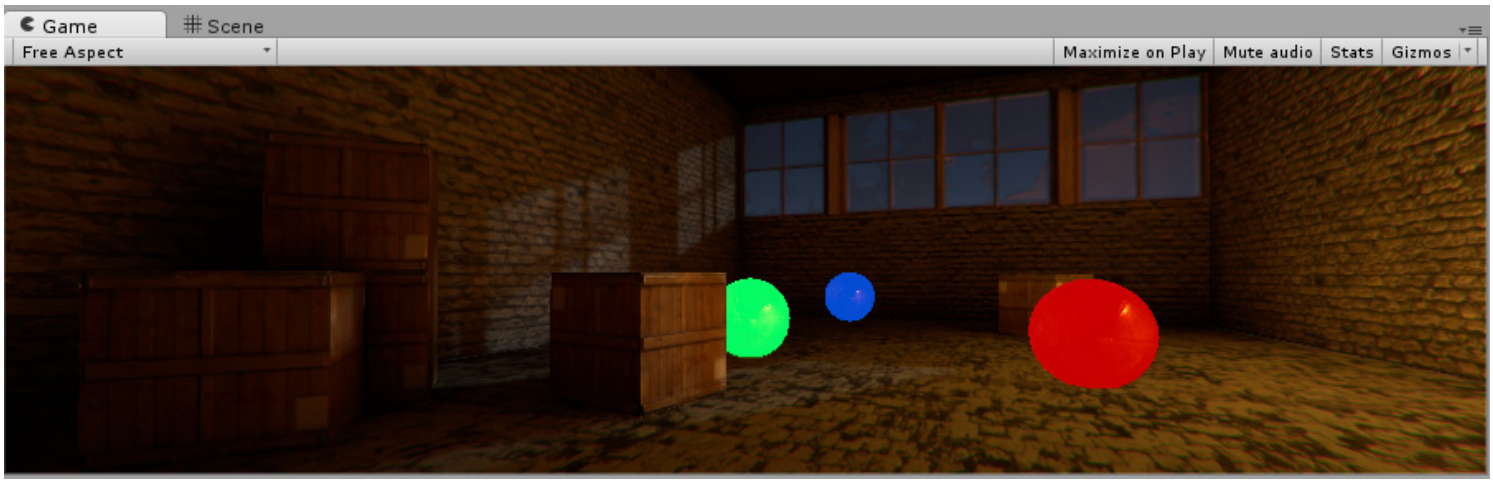
Emissive materials are usually used on GameObjects where some part should appear to be lit up from inside, such as the screen of a monitor, the disc brakes of a car braking at high speed, glowing buttons on a control panel, or a monster's eyes which are visible in the dark.

You can define basic emissive materials with a single color and emission level. To enable the **Emission** property, tick the **Emission** checkbox. This enables the **Color** and **Global Illumination** sub-properties. Click the **Color** box to open the **HDR Color** panel. Here you can alter the color of the illumination and the intensity of the emission:



A material with an orange emission color, and an emission brightness of 0.5

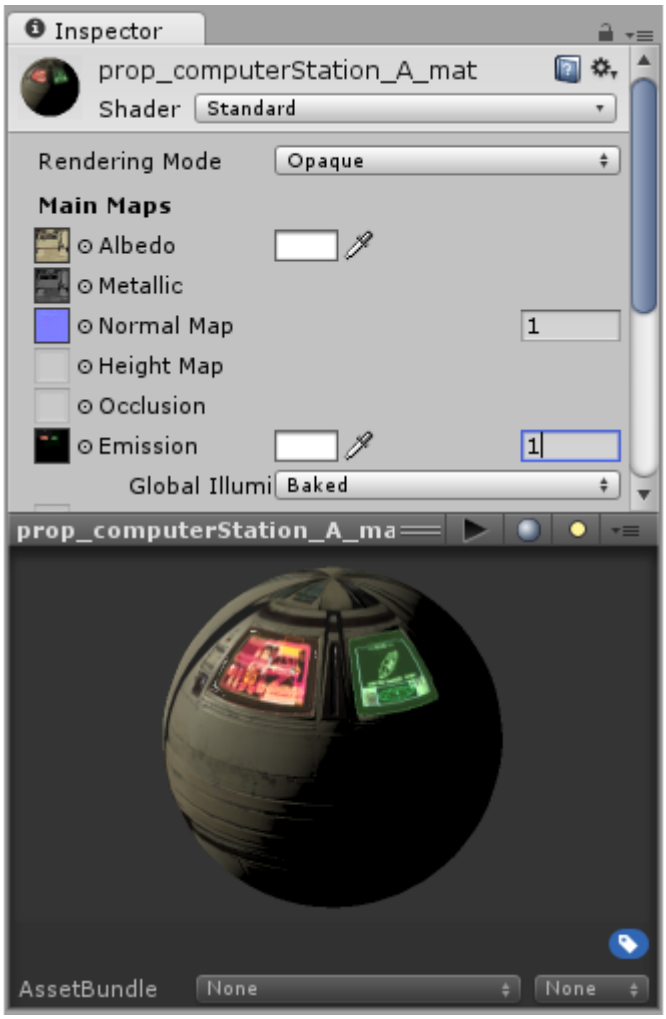
GameObjects that use these materials appear to remain bright even in dark areas of your Scene.



Red, Green and Blue spheres using emissive materials. Even though they are in a dark Scene, they appear to be lit from an internal light source.

As well as simple control over emission using a flat color and brightness setting, you can assign an emission map to this parameter. As with the other texture map parameters, this gives you much finer control over which areas of your material appear to emit light.

If a texture map is assigned, the full color values of the texture are used for the emission color and brightness. The emission value numeric field remains, which you can use as a multiplier to boost or reduce the overall emission level of your material.



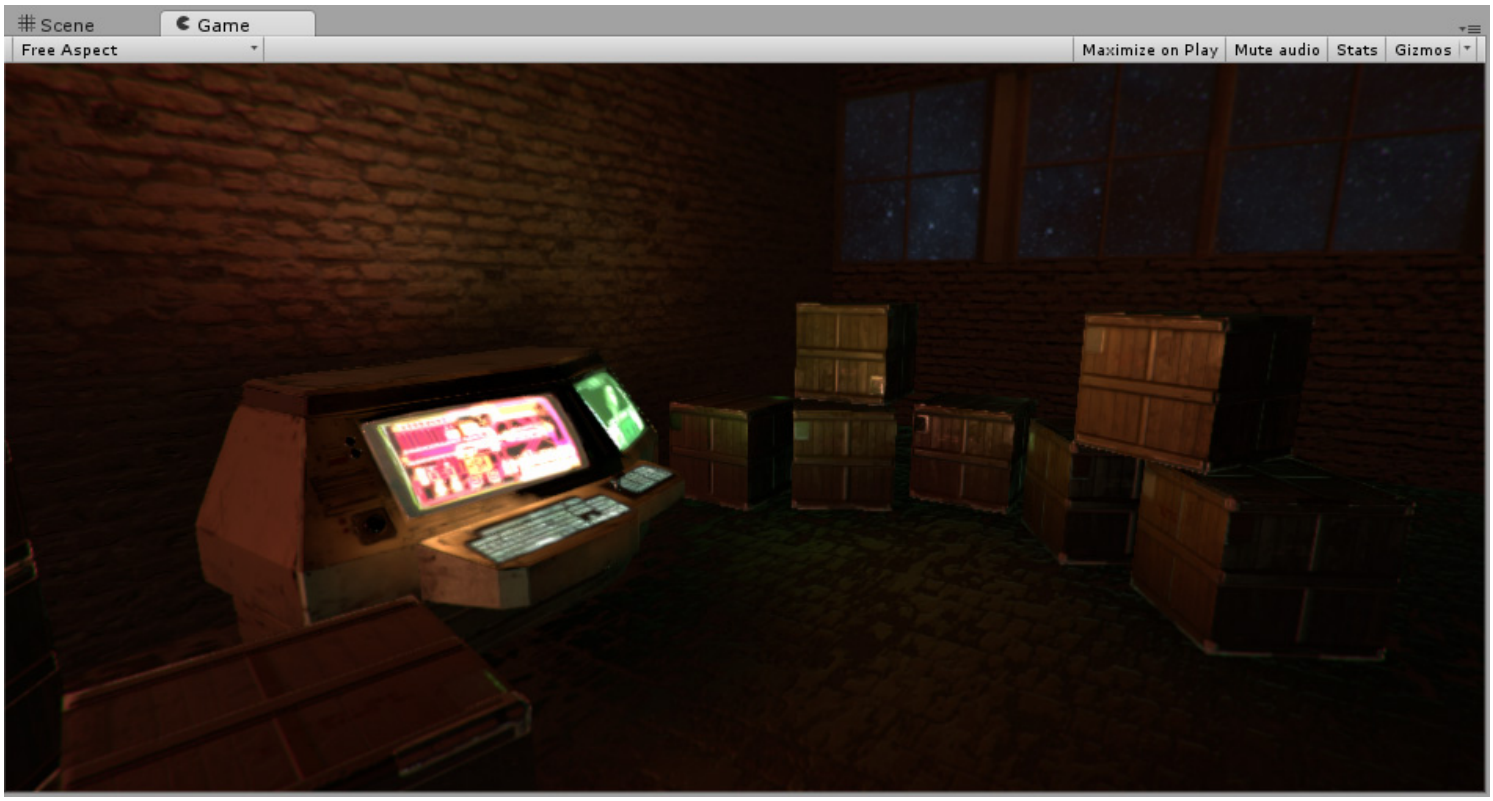
Shown in the inspector, Left: An emission map for a computer terminal. It has two glowing screens and glowing keys on a keyboard. Right: The emissive material using the emission map. The material has both emissive and non-emissive areas.

In this image, there are areas of high and low levels of light, and shadows falling across the emissive areas which gives a full representation of how emissive materials look under varying light conditions.

As well as the emission color and brightness, the Emission parameter has a Global Illumination setting, allowing you to specify how the apparent light emitted from this material affects the contextual lighting of other nearby GameObjects. There are two options

Realtime - Unity adds the emissive light from this Material to the **Realtime** Global Illumination calculations for the Scene. This means that this emissive light affects the illumination of nearby GameObjects, including ones that are moving.

Baked - The emissive light from this material is baked into the static **lightmaps** for the Scene, so other nearby static GameObjects appear to be lit by this material, but dynamic GameObjects are affected.



Baked emissive values from the computer terminal's emission map light up the surrounding areas in this dark Scene

---

2018–08–29 Page amended with limited editorial review

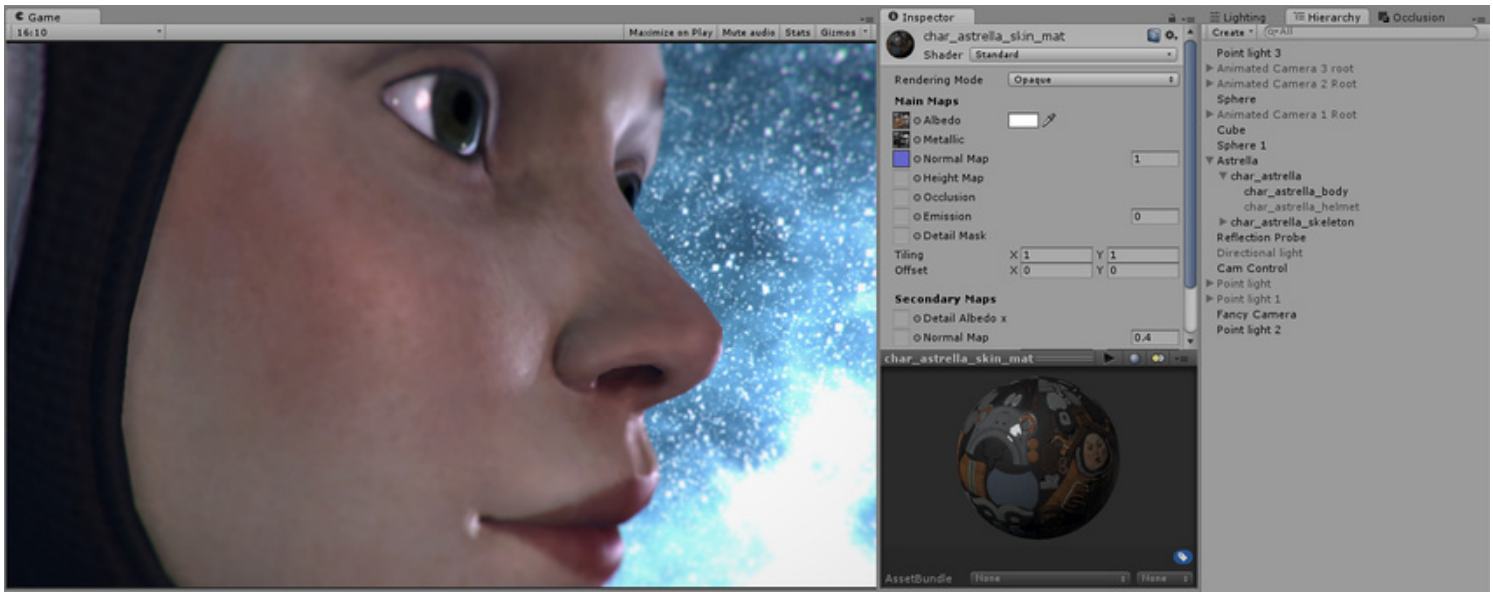
Secondary Maps (Detail Maps) & Detail Mask

Leave feedback

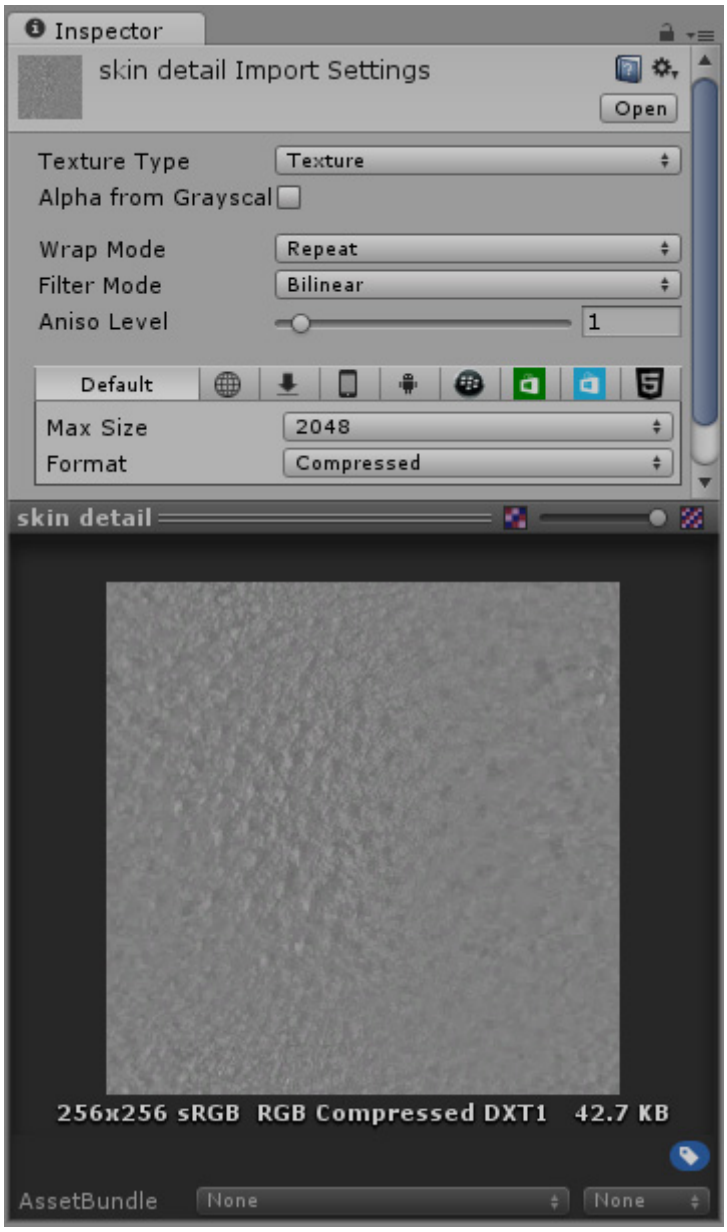
Secondary Maps (or Detail maps) allow you to overlay a second set of textures on top of the main textures listed above. You can apply a second Albedo colour map, and a second Normal map. Typically, these would be mapped on a much smaller scale repeated many times across the object's surface, compared with the main Albedo and Detail maps.

The reason for this is to allow the material to have sharp detail when viewed up close, while also having a normal level of detail when viewed from further away, without having to use a single extremely high texture map to achieve both goals.

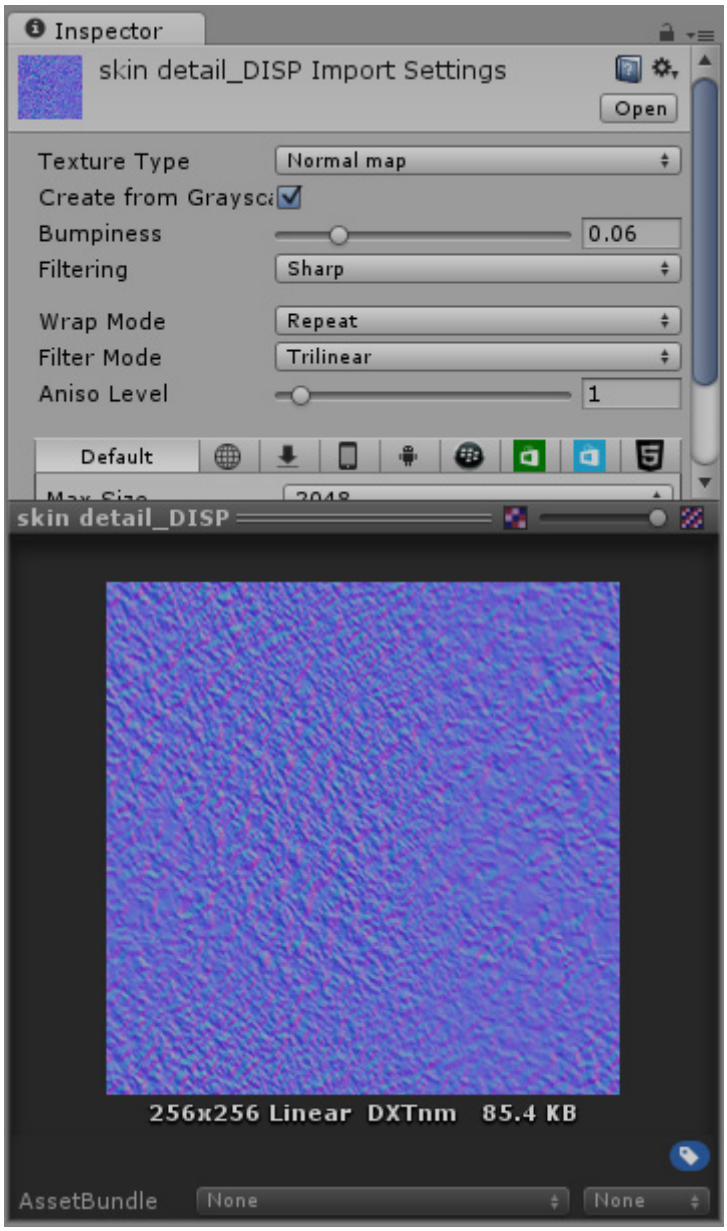
Typical uses for detail textures would be: - Adding skin detail, such as pores and hairs, to a character's skin - Adding tiny cracks and lichen growth to a brick wall - adding small scratches and scuffs to a large metal container



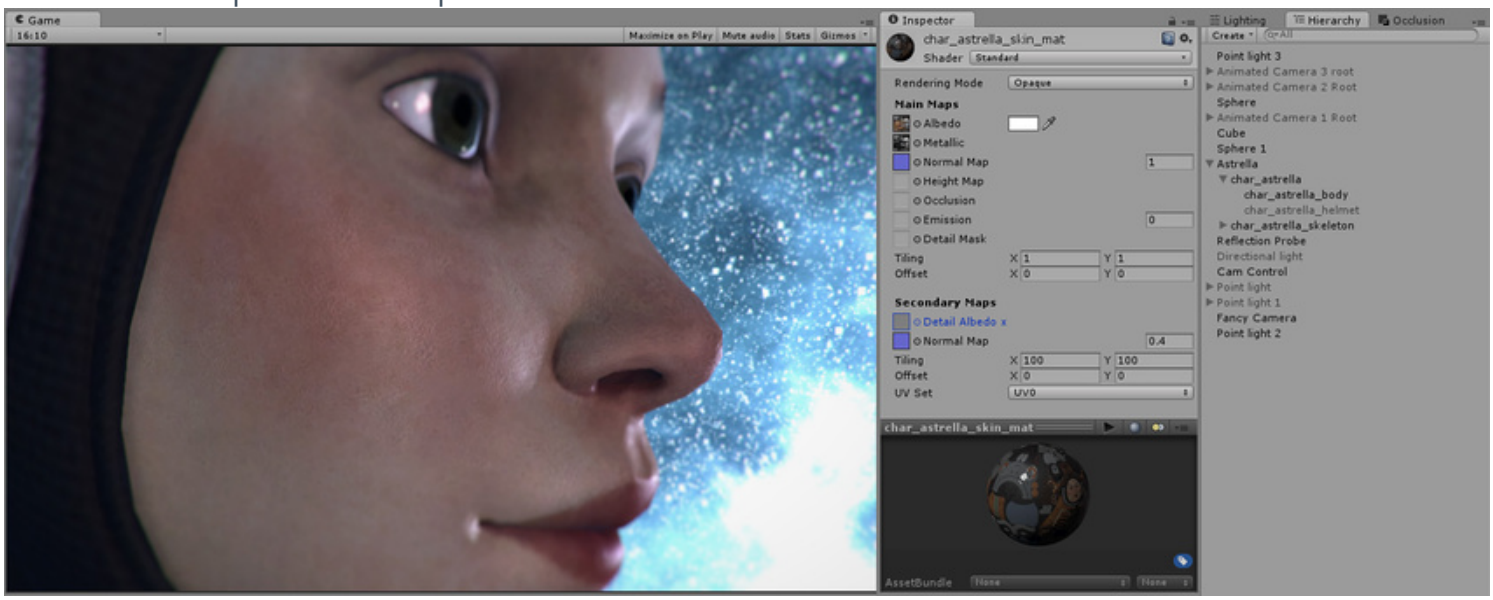
This character has a skin texture map, but no detail texture yet. We will add skin pores as a detail texture.



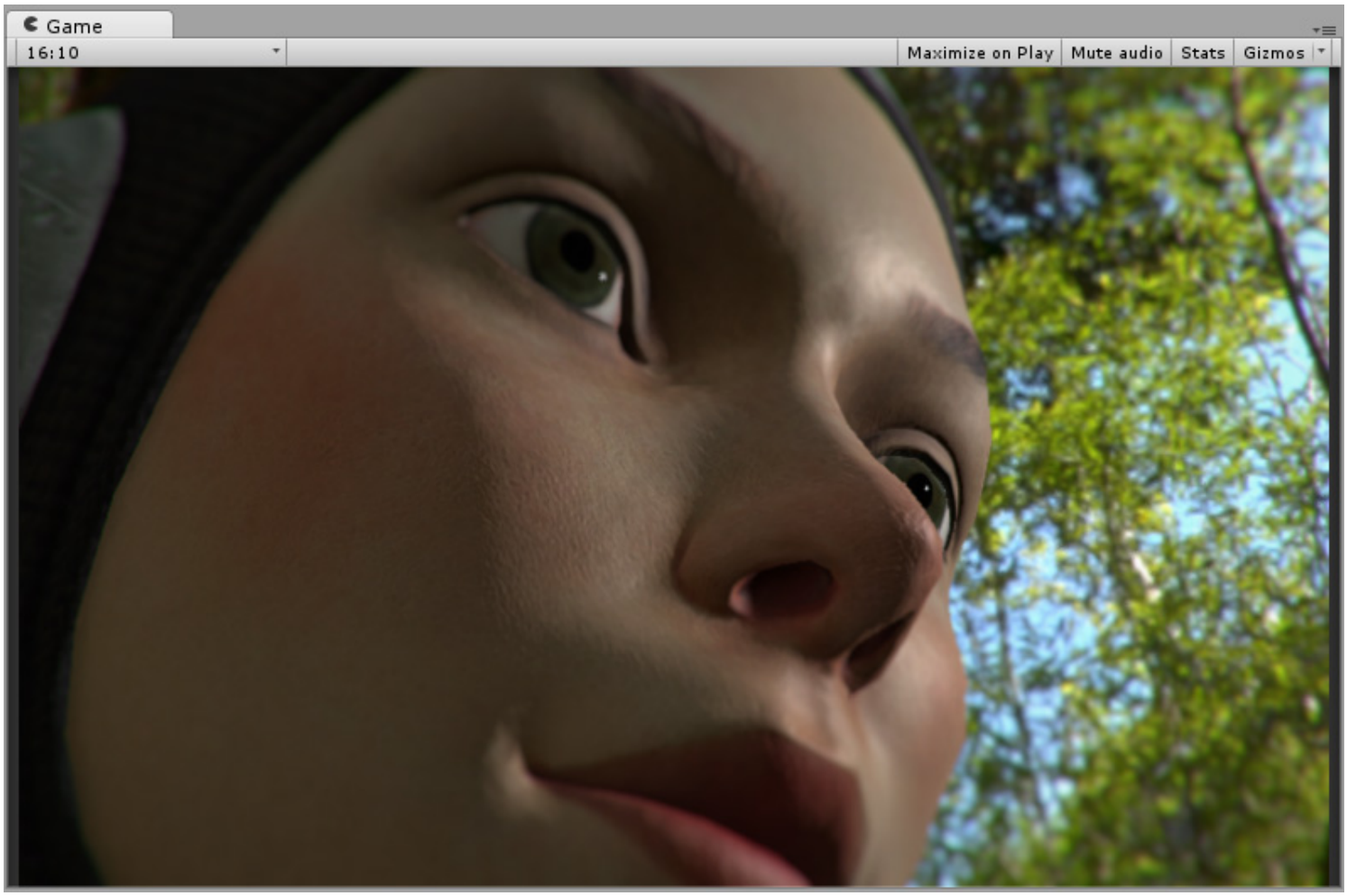
The Albedo skin pore detail texture



The normal map for the skin pore detail



The end result, the character now has subtle skin pore detail across her skin, at a much higher resolution than the base Albedo or Normal map layer would have allowed.



Detail textures can have a subtle but striking effect on the way light hits a surface. This is the same character in a different lighting context.

If you use a single normal map do ALWAYS plug it into the primary channel. The Secondary normal map channel is more expensive than the primary one but has the exact same effect.

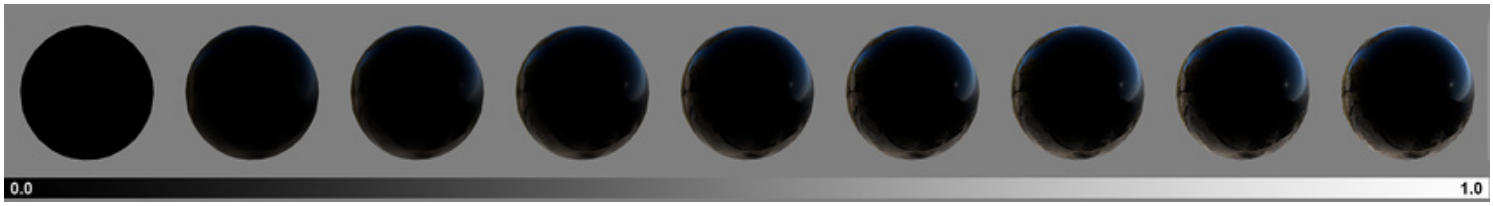
Detail Mask

The detail mask texture allows you to mask off certain areas of your model to have the detail texture applied. This means you can show the detail texture in certain areas, and hide it in others. In the example of the skin pores above, you might want to create a mask so the pores are not shown on the lips or eyebrows.

The Fresnel Effect

Leave feedback

One important visual cue of objects in the real world has to do with how they become more reflective at grazing angles (illustrated below). This is called the **Fresnel effect**.



The fresnel effect visible at grazing angles in relation to the viewer is increasingly apparent as the surface of a material becomes smoother

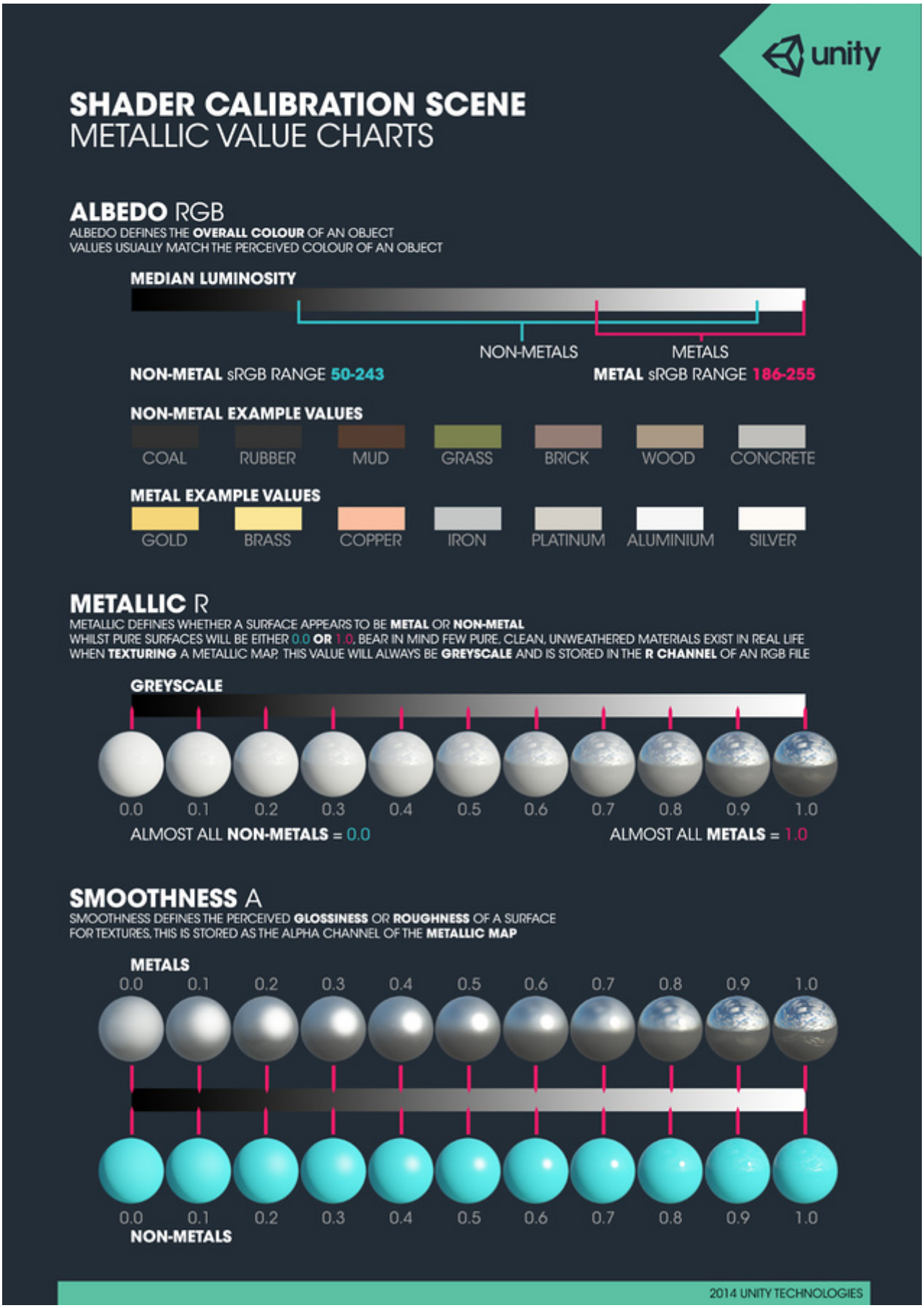
There are two things to note in this example; firstly, these reflections only appear around the edges of the sphere (that's when its surface is at a grazing angle), and also that they become more visible and sharper as the smoothness of the material goes up.

In the **Standard shader** there is no direct control over the Fresnel effect. Instead it is indirectly controlled through the smoothness of the material. Smooth surfaces will present a stronger Fresnel, totally rough surfaces will have no Fresnel.

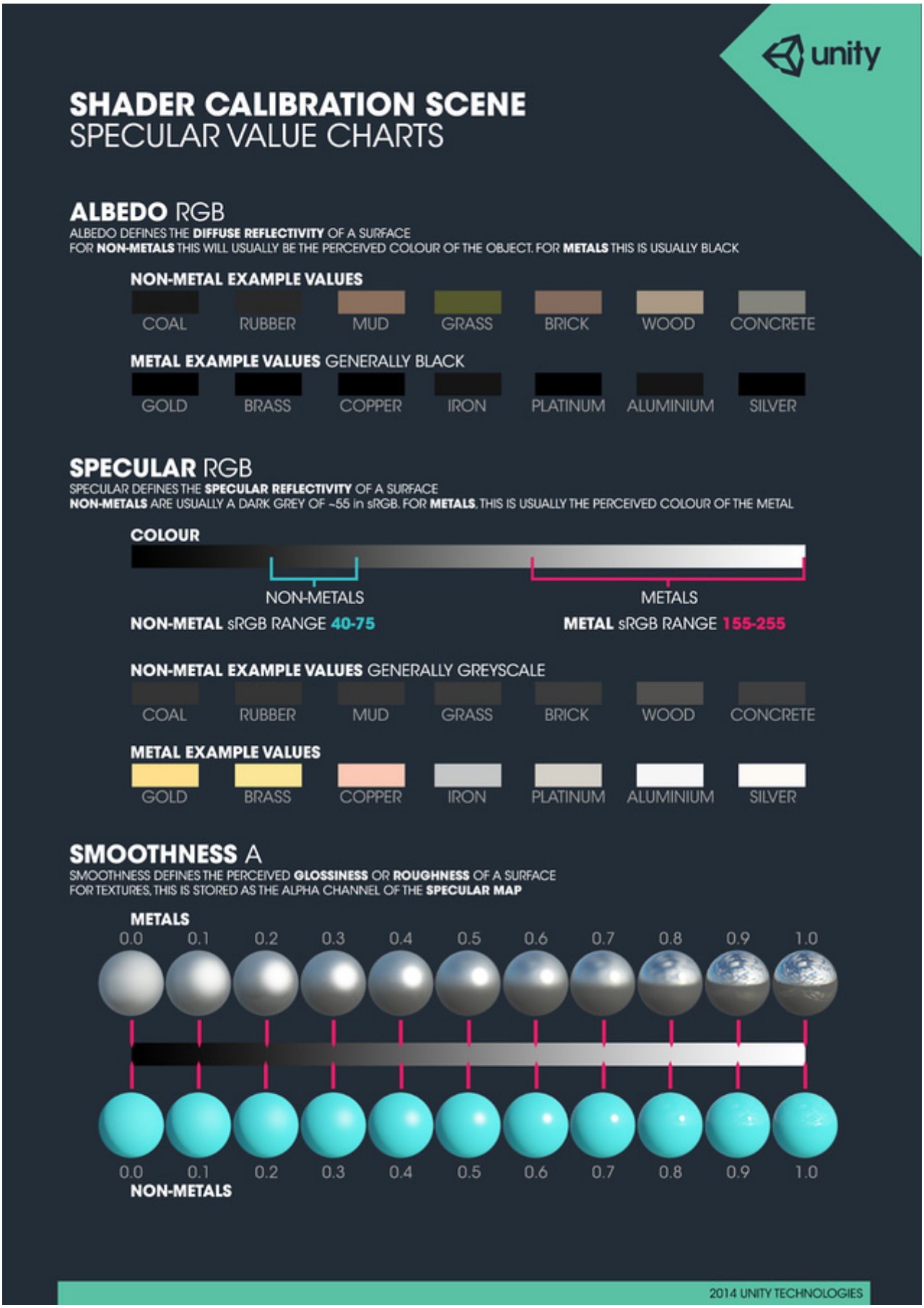
Material charts

Leave feedback

Use these charts as reference for realistic settings:



Reference Chart for Metallic settings



Reference Chart for Specular settings

There are also hints on how to make realistic materials in these charts. In essence it is about choosing a workflow (default or metallic) and obtaining relevant values for your maps or colour pickers. For instance, if we wanted to

make shiny white plastic, we would want a white Albedo. Since it is not a metal we would want a dark Specular (or a very low Metallic value) and finally a very high Smoothness.

Make your own

Leave feedback

The **Standard shader** is a solid example of a **shader** using the new PBS System and should be valuable for a range of uses, but you can of course build on this editing the **shader** and bringing more esoteric materials and workflows to suit your projects.

The shader code source is provided here.

Standard Particle Shaders

Leave feedback

The Unity Standard Particle Shaders are built-in shaders that enable you to render a variety of **Particle System** effects. These shaders provide various particle-specific features that aren't available with the Standard **Shader**.

To use a Particle Shader:

Select the Material you want to apply the shader to. For example, you could apply a Flame Material to a Fire Particle System effect.

In the Material's **Inspector**, select **Shader Particles**.

Choose the Particle Shader that you want to use, such as **Standard Surface**.

Enable and disable the various Particle Shader properties in the Inspector.

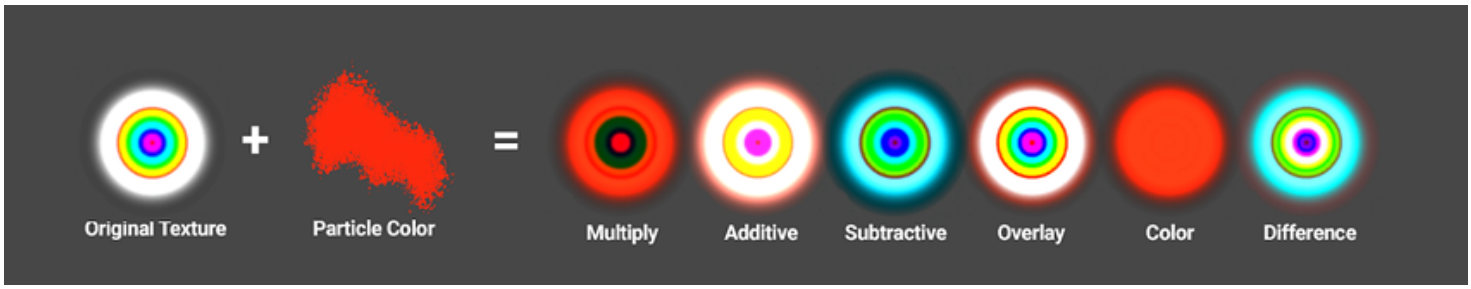
Properties

The Standard Particle Shaders have the same set of properties as the Standard Shader (or a subset of those properties, depending on the Shader). This page describes the properties and options that are additional to the Standard Shader properties. For information on the Standard Shader properties, see documentation on Material parameters.

Blending Options

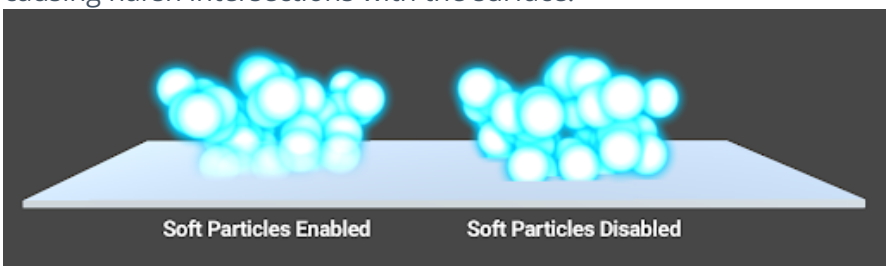
All of the Standard Particle Shaders have **Blending Options** that enable you to blend particles with the objects surrounding them in different ways.

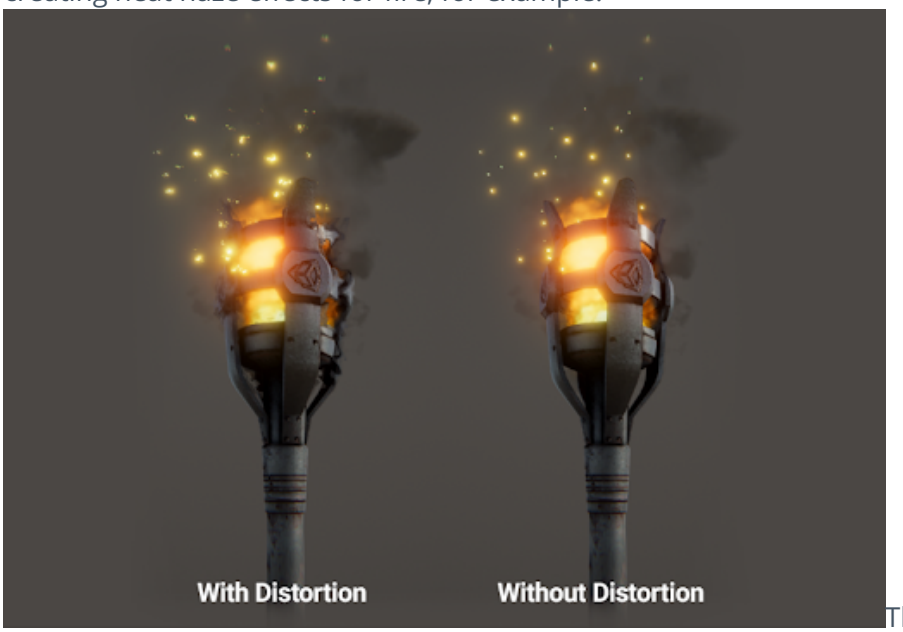
Property:	Function:
Rendering Mode	The Standard Particle Shaders can have the following additional Rendering Mode options: **Additive**: Adds the background and particle colors together. This is useful for glow effects, like those you might use for fire or magic spells. **Subtractive**: Subtracts particle colors from the background, which darkens the particles against the background. This is useful for foggy effects, like those you might use for steam, or thick black smoke. **Modulate**: Multiplies the material pixels with the background color. This is useful for portals and light-rays.
Color Mode	Control how the albedo texture is combined with the particle color. The **Color Mode** options are: **Multiply**: Multiplies the particle color with the particle texture. **Additive**: Preserves a hot spot, such as a white part of the particle texture, while adding the particle color to the darker pixels of the texture. **Subtractive**: Subtracts the particle color from the particle texture. **Overlay**: Gives more contrast to the original color and adds the particle color to the gray values. This is similar to Additive, but preserves the original colors. **Color**: Uses the alpha channel from the particle texture and the color from the particle itself. This is useful for overwriting particles with the same color, while keeping their original "shape". **Difference**: Subtracts the particle color from the texture, or the texture from the color, to get a positive value. This is useful for a range of effects where you want a more dynamic color change. See image below table for a demonstration of this effect.



Color Modes allow for different ways of combining the particle color with the albedo texture

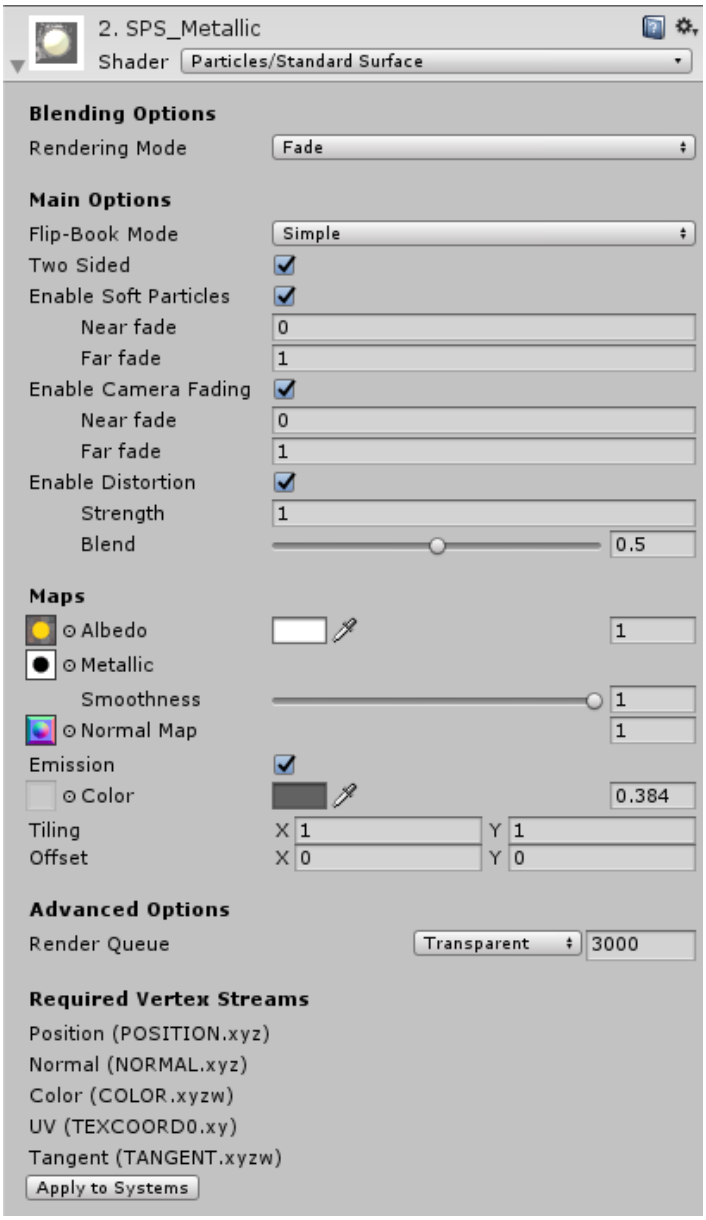
Main Options

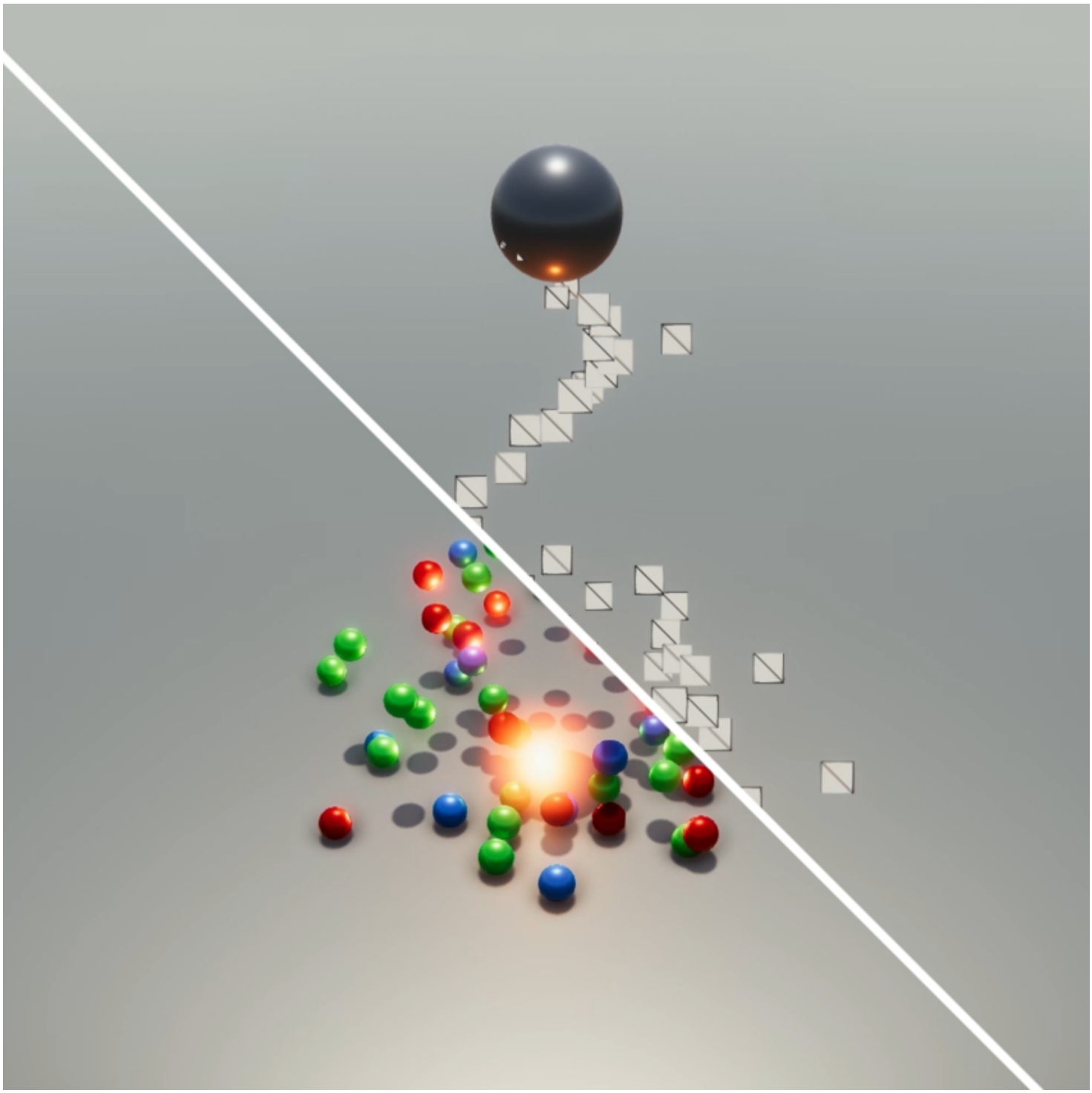
Property	Function
Flip-Book Mode	Render flip-books as individual frames or blend the frames together to give smoother animations. Set to either: **Simple** - Render frames in a flip-book as a sequence of individual frames. **Blended** - Blend the frames in a flip-book to render the flip-book as a smooth animation.
Two Sided	Render both the front and back faces of the particle. When disabled, Unity only renders the front face of the geometry, which is the face in the **camera**'s view.
Enable Soft Particles	Fade out particles when they get close to the surface of objects written into the depth buffer. This is useful for avoiding hard edges when particles intersect with opaque geometry. For example, by enabling soft particles, you can make the particle system emit particles close to an opaque surface without causing harsh intersections with the surface: 
Enable Camera Fading	Fade out particles when they get close to the camera. Set to: **Near fade** - The closest distance particles can get to the camera before they fade from the camera's view. **Far fade** - The farthest distance particles can get away from the camera before they fade from the camera's view.

Property	Function
Enable Distortion	Make particles perform fake refraction with the objects drawn before them. Distortion is ideal for creating heat haze effects for fire, for example:  This effect can be quite expensive because it captures the current frame to a texture.

Standard Particles Surface Shader

This shader comes with functionality similar to the Standard Shader, but works especially well with particles. Like the Standard Shader, it supports **Physically Based Shading**. It does not include features that are unsuitable for dynamic particles, such as lightmapping.

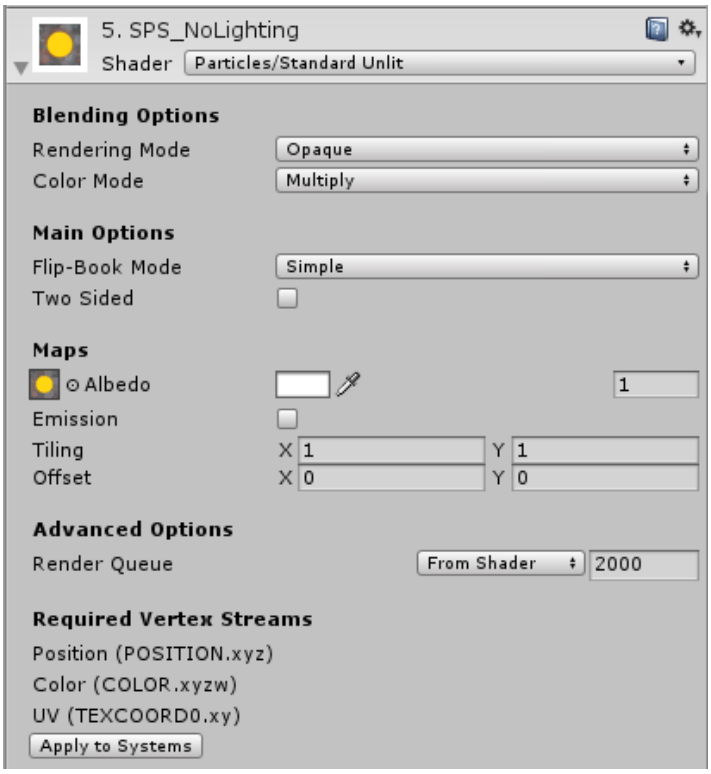

2. SPS_Metallic
Shader
Particles/Standard Surface
Blending Options
Rendering Mode
Fade
Main Options
Flip-Book Mode
Simple
Two Sided
Enable Soft Particles
Near fade
0
Far fade
1
Enable Camera Fading
Near fade
0
Far fade
1
Enable Distortion
Strength
1
Blend
0.5
Maps
Albedo
1
Metallic
Smoothness
1
Normal Map
1
Emission
Color
0.384
Tiling
X 1
Y 1
Offset
X 0
Y 0
Advanced Options
Render Queue
Transparent
3000
Required Vertex Streams
Position (POSITION.xyz)
Normal (NORMAL.xyz)
Color (COLOR.xyzw)
UV (TEXCOORD0.xy)
Tangent (TANGENT.xyzw)
Apply to Systems



An example of billboard particles using the Standard Particle Surface Shader with a normal map

Standard Particles Unlit Shader

This simple shader is faster than the **Surface Shader**. It supports all of the generic particle controls, such as flipbook blending and distortion, but does not perform any lighting calculations.



2017–10–16 Page published with no editorial review

Standard Particle Shaders added in 2017.3

Physically Based Rendering Material Validator

Leave feedback

The Physically Based **Rendering** Material Validator is a draw mode in the **Scene View**. It allows you to make sure your materials use values which fall within the recommended reference values for physically-based **shaders**. If pixel values in a particular Material fall outside of the reference ranges, the Material Validator highlights the **pixels** in different colors to indicate failure.

To use the Material Validator, select the **Scene** View's **draw mode** drop-down menu, which is is usually set to **Shaded** by default.

The scene view's draw mode drop-down menu

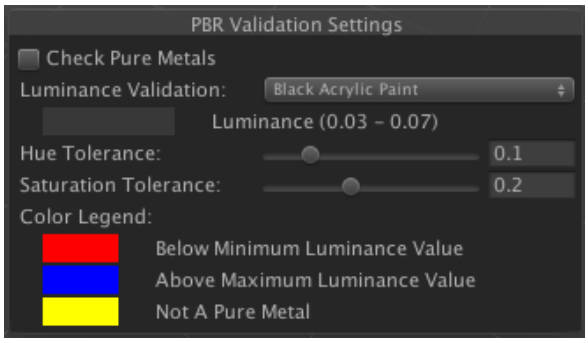
Navigate to the **Material Validation** section. The Material Validator has two modes: **Validate Albedo** and **Validate Metal Specular**.

The material validation options in the scene view draw mode drop-down menu

Note: You can also check the recommended values with Unity's Material Charts. You still need to use these charts when authoring Materials to decide your **albedo** and **metal specular** values. However, the Material Validator provides you with a visual, in-editor way of quickly checking whether your Materials' values are valid once your **Assets** are in the Scene.

Also note: The validator only works in Linear color space. Physically Based Rendering is not intended for use with Gamma color space, so if you are using Physically Based Rendering and the PBR Material Validator, you should also be using Linear color space.

Validate Albedo mode



The PBR Validation Settings when in Validate Albedo mode, which appear in the scene view

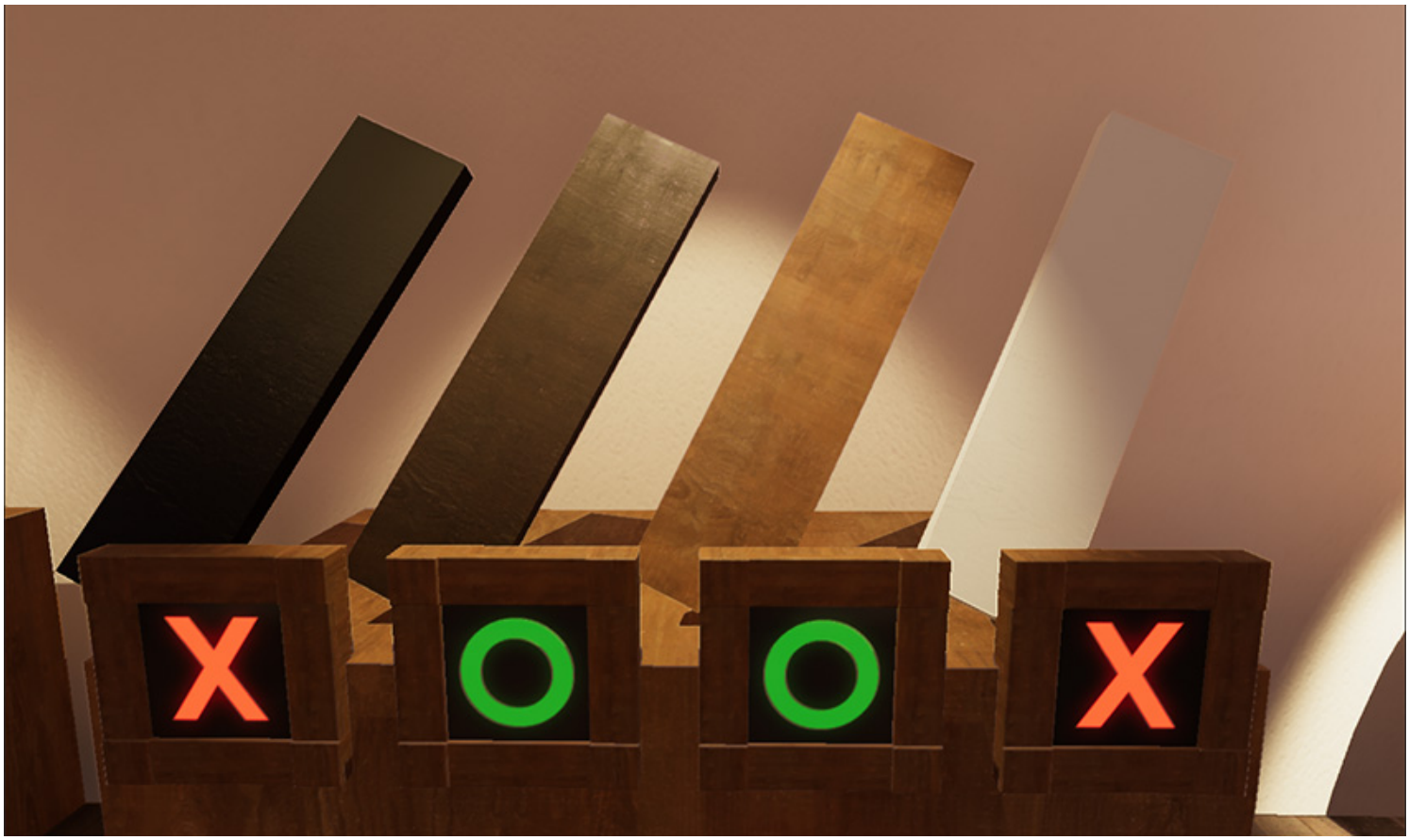
The PBR Validation Settings that appear in the Scene view when you set Material Validation to Validate Albedo.

Property:	Function:
Check Pure Metals	Enable this checkbox if you want the Material Validator to highlight in yellow any pixels it finds which Unity defines as metallic, but which have a non-zero albedo value. See Pure Metals, below, for more details. By default, this is not enabled.
Luminance Validation	Use the drop-down to select a preset configuration for the Material Validator. If you select any option other than **Default Luminance**, you can also adjust the Hue Tolerance and Saturation Tolerance. The color bar underneath the name of the property indicates the albedo color of the configuration. The Luminance value underneath the drop-down indicates the minimum and maximum luminance value. The Material Validator highlights any pixels with a luminance value outside of these values. This is set to **Default Luminance** by default.
Hue Tolerance	When checking the albedo color values of your material, this slider allows you to control the amount of error allowed between the hue of your material, and the hue in the validation configuration.

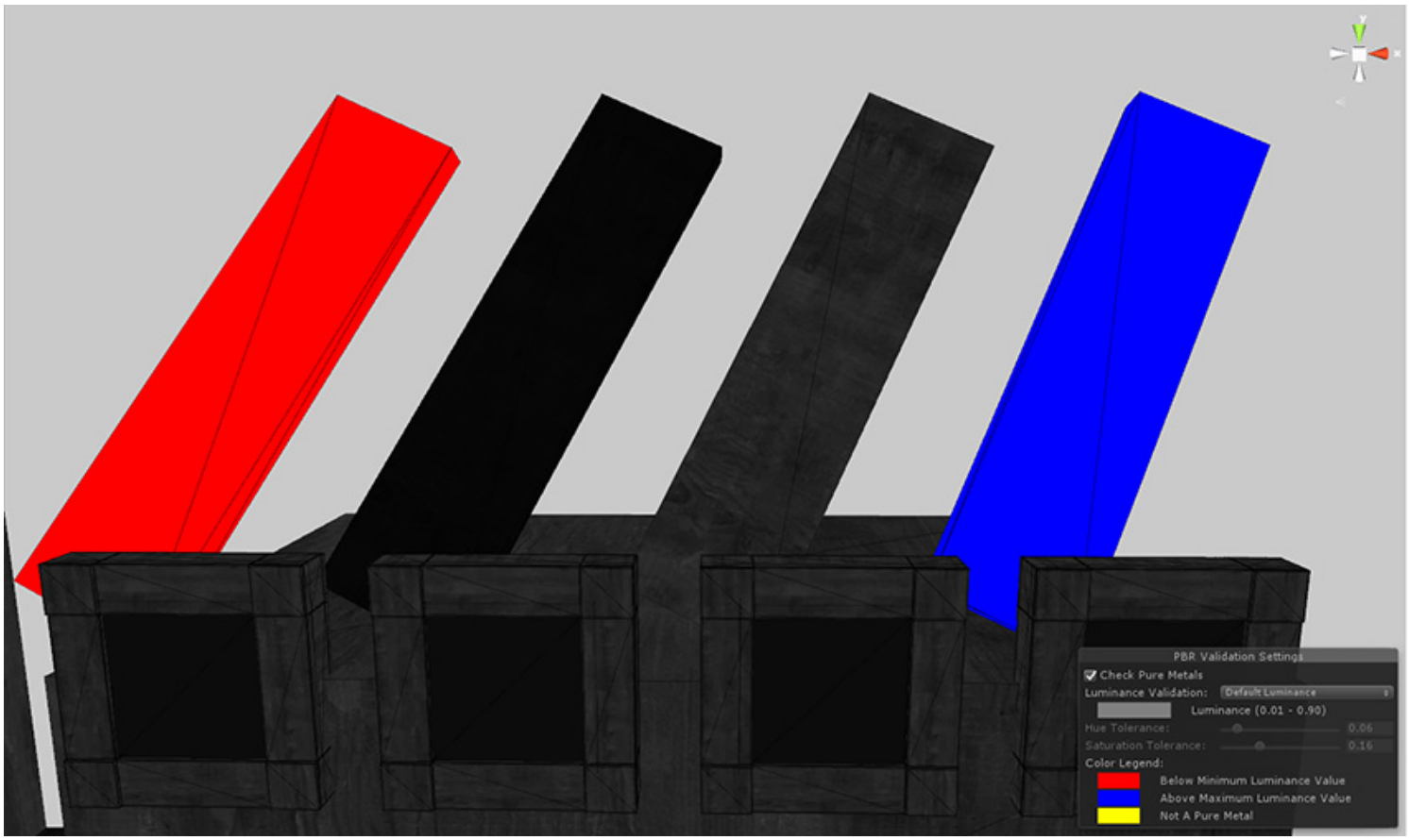
Property:	Function:
Saturation Tolerance	When checking the albedo color values of your material, this slider allows you to control the amount of error allowed between the saturation of your material, and the saturation in the validation configuration.
Color Legend	These colors correspond to the colours that the Material Validator displays in the Scene view when the pixels for that Material are outside the defined values.
Red Below Minimum Luminance Value	The Material Validator highlights in red any pixels which are below the minimum luminance value defined in **Luminance Validation** (meaning that they are too dark).
Blue Above Maximum Luminance Value	The Material Validator highlights in blue any pixels which are above the maximum luminance value defined in **Luminance Validation** (meaning that they are too bright).
Yellow Not A Pure Metal	If you have Check Pure Metals enabled, the Material Validator highlights in yellow any pixels which Unity defines as metallic, but which have a non-zero albedo value. See Pure Metals, below, for more details.

Unity's Material charts define the standard luminance range as 50–243 sRGB for non-metals, and 186–255 sRGB for metals. **Validate Albedo** mode colors any pixels outside of these ranges with different colors to indicate that the value is too low or too high.

In the example below, the first texture is below the minimum luminance value, and therefore too dark. The fourth texture is above the minimum luminance value, and therefore too bright.



A Scene (without the Material Validator enabled) in which the first and fourth Materials have incorrect albedo values



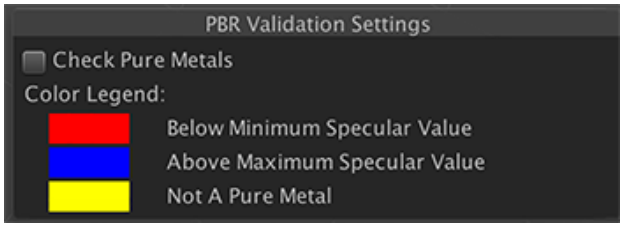
The same Scene with the Material Validator enabled and set to Validate Albedo. The texture that is below the minimum luminance value is red. The texture that is above the minimum luminance value is blue

The material charts provide albedo values for common Materials. The brightness of albedo values has a dramatic impact on the amount of diffuse bounce light generated, so it is important for Global Illumination baking to make sure that your different Material types are within the correct luminance ranges, in proportion with each other. To help you get these values right, you can select from the presets in the Luminance Validation drop-down, which provides common Material albedo values to verify the luminance ranges of particular Material types.

Overriding the default luminance values

Depending on the art style of your project, you might want the luminance values of Materials to differ from the preset luminance ranges. In this case, you can override the built-in albedo values used by the Material Validator with your own values. To override the preset luminance ranges, assign an array of AlbedoSwatchInfo values for each desired Material type to the property EditorGraphicsSettings.albedoSwatches.

Validate Metal Specular mode



The PBR validations settings, when in Validate Metal Specular mode

The PBR Validation Settings that appear in the Scene view when you set **Material Validation** to **Validate Metal Specular**.

Property:	*Function:*
Check Pure Metals	Enable this checkbox if you want the Material Validator to highlight in yellow any pixels it finds which Unity defines as metallic, but which have a non-zero albedo value. See Pure Metals, below, for more details. By default, this is not enabled.

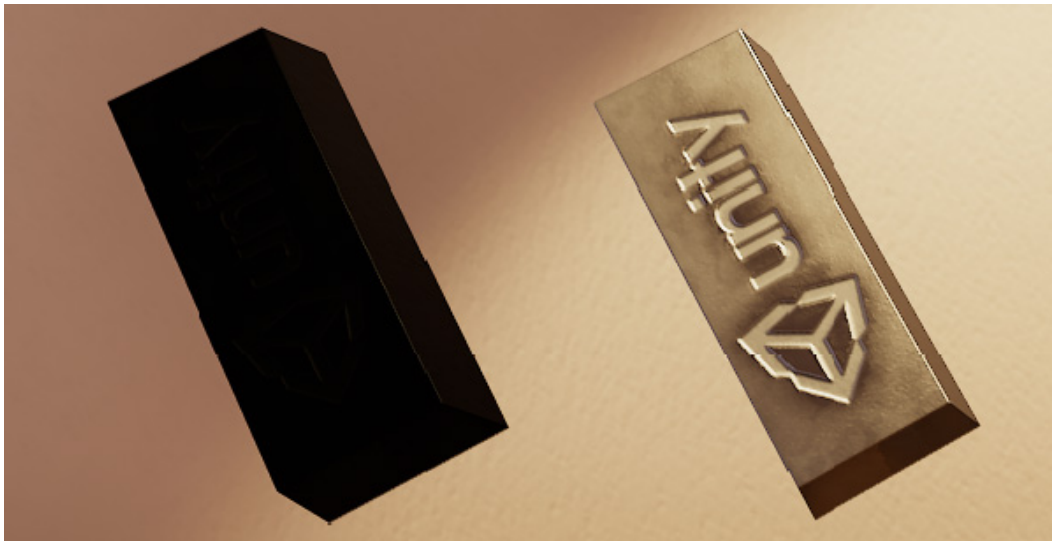
Property:	*Function:*
Color Legend	These colors correspond to the colours that the Material Validator displays in the Scene view when the pixels for that Material are invalide - meaning their specular value falls outside the valid range for that type of material (metallic or non-metallic). See below this table for the valid ranges.
Blue Below Minimum Specular Value	The Material Validator highlights in red any pixels which are below the minimum specular value. (40 for non-metallic, or 155 for metallic).
Red Above Maximum Specular Value	The Material Validator highlights in blue any pixels which are above the maximum specular value. (75 for non-metallic, or 255 for metallic).
Yellow Not A Pure Metal	If you have **Check Pure Metals** enabled, the Material Validator highlights in yellow any pixels which Unity defines as metallic, but which have a non-zero albedo value. See Pure Metals, below, for more details.

Unity's Material charts define two separate specular color ranges:

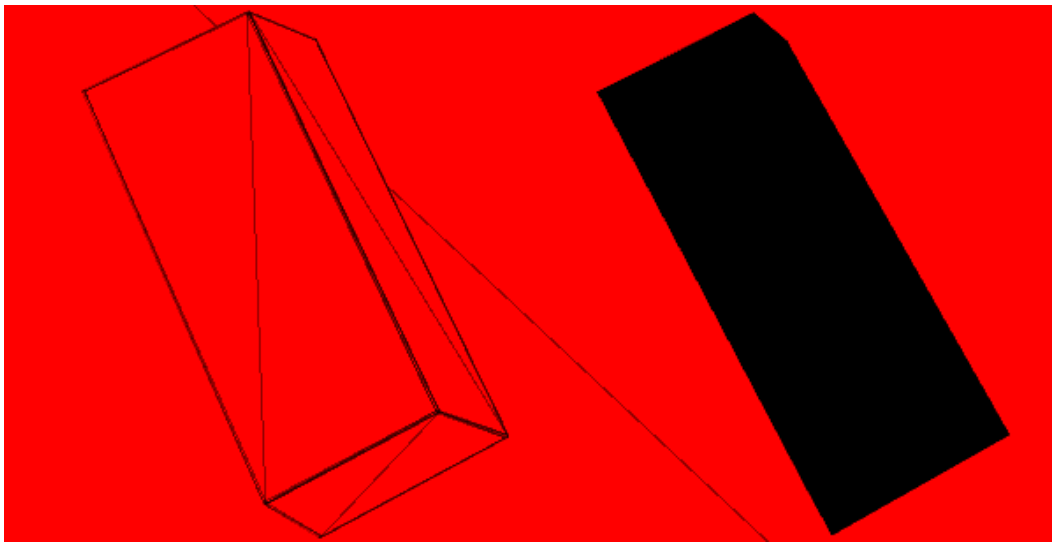
Non-metallic materials: 40–75 sRGB
Metallic materials: 155 - 255 sRGB

In Unity, all non-metallic Materials have a constant specular color that always falls within the correct range. However, it is common for metallic Materials to have specular values that are too low. To help you identify metallic Materials with this issue, the Material Validator's **Validate Metal Specular** mode colors all pixels that have a specular color value that is too low. This includes all non-metallic materials by definition.

In the example below, the left material is below the minimum specular value, and therefore too dark. This also applies to the Scene's background. The right material has specular values with in the valid range.



A Scene with two metallic Materials. The left has incorrect metallic specular values

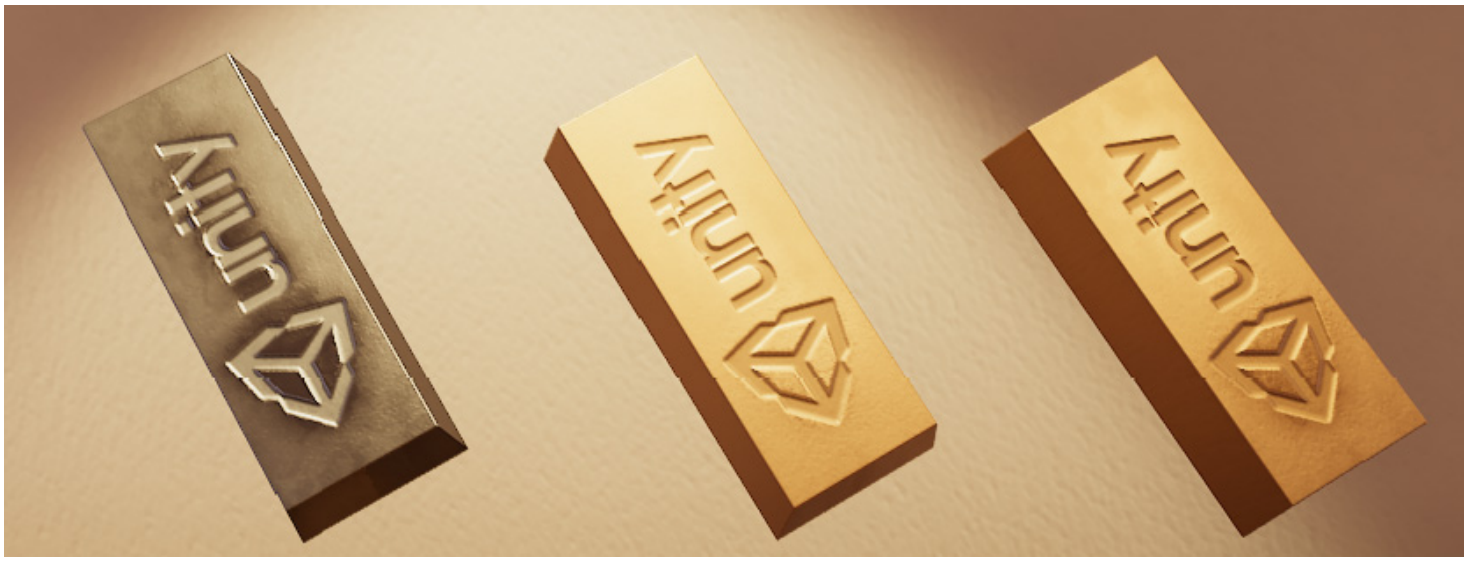


The same Scene with the Material Validator enabled and set to **Validate Metal Specular**

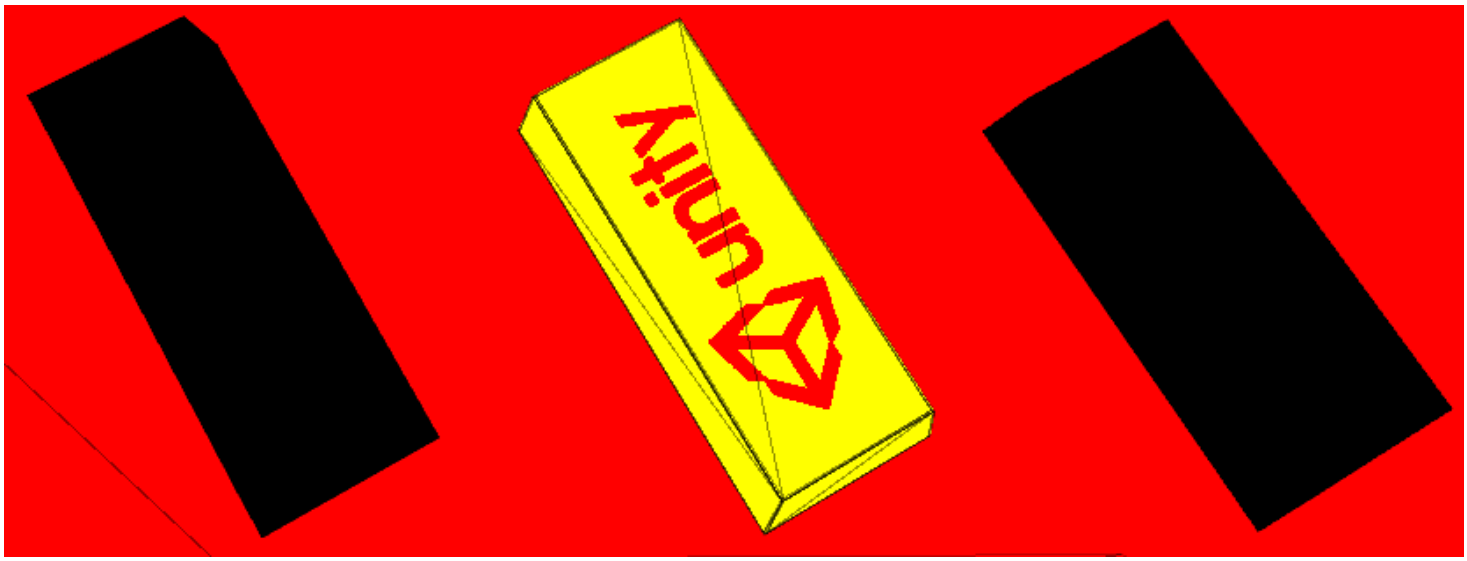
Pure Metals

Unity defines physically-based shading materials with a specular color greater than 155 sRGB as metallic. For Unity to define a metallic Material as a **pure metal**

If a non-metallic surface has a specular color value that is too high, but has a non-zero albedo value, this is often due to an authoring error. The Material Validator also has an option called **Check Pure Metals**. When you enable this option, the Material Validator colors in yellow any Material that Unity defines as metallic but which has a non-zero albedo value. An example of this can be seen in the images below. It shows three materials, the left and right materials are pure metals, but the middle material is not, so the Material Validator colors it yellow:



A Scene with three metallic Materials. The middle Material is not a pure metal (it has a non-zero albedo value)



The same Scene with the **Material Validator** enabled and set to **Validate Metal Specular**, with Check Pure Metals enabled)

In the second image above, the background is red because the Materials in the background are below the minimum specular value for the Material Validator's **Validate Metal Specular** mode.

For complex materials that combine metallic and non-metallic properties, the pure metal checker is likely to pick up some invalid pixels, but if a Material is totally invalid, it's usually a sign of an authoring error.

Implementation

The Material Validator works with any Materials that use Unity's Standard shader or surface shaders. However, custom shaders require a pass named "`META`". Most custom shaders that support lightmapping already have this pass defined. See documentation on Meta pass for more details.

Carry out the following steps to make your custom shader compatible with the Material Validator:

Add the following pragma to the meta pass: `#pragma shader_feature EDITOR_VISUALIZATION`

In the `UnityMetaInput` structure, assign the specular color of the Material to the field called `SpecularColor`, as shown in the code example below.

Here is an example of a custom meta pass:

```
Pass
{
    Name "META"
    Tags { "LightMode"="Meta" }

    Cull Off

    CGPROGRAM
    #pragma vertex vert_meta
    #pragma fragment frag_meta

    #pragma shader_feature _EMISSION
    #pragma shader_feature _METALLICGLOSSMAP
    #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
    #pragma shader_feature ___ _DETAIL_MULX2
    #pragma shader_feature EDITOR_VISUALIZATION
```

```
    float4 frag_meta(v2f_meta i) : SV_TARGET
    {
        UnityMetaInput input;
        UNITY_INITIALIZE_OUTPUT(UnityMetaInput, input);
        float4 materialSpecularColor = float4(1.0f, 0.0f, 0.0f, 1.0f);
        float4 materialAlbedo = float4(0.0f, 1.0f, 0.0f, 1.0f);
        input.SpecularColor = materialSpecularColor;
        input.Albedo = materialAlbedo;

        return UnityMetaFragment(input);
    }
}
```

2018–03–28 Page published with editorial review
Material Validator updated in Unity 2017.3

Accessing and Modifying Material parameters via script

Leave feedback

All the parameters of a Material that you see in the **inspector** when viewing a material are accessible via script, giving you the power to change or animate how a material works at runtime.

This allows you to modify numeric values on the Material, change colours, and swap textures dynamically during gameplay. Some of the most commonly used functions to do this are:

Function Name	Use
SetColor	Change the color of a material (Eg. the albedo tint color)
SetFloat	Set a floating point value (Eg. the normal map multiplier)
SetInt	Set an integer value in the material
SetTexture	Assign a new texture to the material

The full set of functions available for manipulating materials via script can be found on the Material class scripting reference.

One important note is that these functions **only set properties that are available for the current shader** on the material. This means that if you have a **shader** that doesn't use any textures, or if you have no **shader** bound at all, calling SetTexture will have no effect. This is true even if you later set a shader that needs the texture. For this reason it is recommended to set the shader you want before setting any properties, however once you've done that you can switch from one shader to another that use the same textures or properties and values will be preserved.

These functions work as you would expect for all *simple* shaders such as the legacy shaders, and the built-in shaders *other than the Standard Shader* (for example, the particle, sprite, UI and unlit shaders). For a material using the *Standard Shader* however, there are some further requirements which you must be aware of before being able to fully modify the Material.

Special requirements for Scripting with the Standard Shader

The Standard Shader has some extra requirements if you want to modify Materials at runtime, because - behind the **scenes** - it is actually many different shaders rolled into one.

These different types of shader are called Shader Variants and can be thought of as all the different possible combinations of the shader's features, when activated or not activated.

For example, if you choose to assign a Normal Map to your material, you activate that *variant* of the shader which supports Normal Mapping. If you subsequently also assign a Height Map then you activate the variant of the shader which supports Normal Mapping and Height Mapping.

This is a good system, because it means that if you use the Standard Shader, but do not use a Normal Map in a certain Material, you are not incurring the performance cost of running the Normal Map shader code - because you are running a variant of the shader with that code omitted. It also means that if you never use a certain feature combination (such as HeightMap & Emissive together), that variant is completely omitted from your build - and in practice you will typically only use a very small number of the possible variants of the Standard Shader.

Unity avoids simply including every possible shader variant in your build, because this would be a very large number, some tens of thousands! This high number is a result not only of each possible combination of features available in the material inspector, but also there are variants of each feature combination for differing **rendering** scenarios such as whether or not **HDR** is being used, **lightmaps**, GI, fog, **etc**. Including all of these would cause slow loading, high memory consumption, and increase your build size and build time.

Instead, Unity tracks which variants you've used by examining the material **assets** used in your project. Whichever variants of the Standard Shader you have included in your project, those are the variants which are included in the build.

This presents two separate problems when accessing materials via script that use the Standard Shader.

1. You must enable the correct Keywords for your required Standard Shader variant

If you use scripting to change a Material that would cause it to use a different variant of the Standard Shader, you must **enable that variant by using the EnableKeyword function**. A different variant would be required if you start using a shader feature that was not initially in use by the material. For example assigning a Normal Map to a Material that did not have one, or setting the Emissive level to a value greater than zero, when it was previously zero.

The specific Keywords required to enable the Standard Shader features are as follows:

Keyword	Feature
_NORMALMAP	Normal Mapping
_ALPHATEST_ON	"Cut out" Transparency **Rendering Mode**
_ALPHABLEND_ON	"Fade" Transparency Rendering Mode
_ALPHAPREMULTIPLY_ON	"Transparent" Transparency Rendering Mode
_EMISSION	Emission Colour or Emission Mapping
_PARALLAXMAP	Height Mapping
_DETAIL_MULX2	Secondary "Detail" Maps (Albedo & Normal Map)
_METALLICGLOSSMAP	Metallic/Smoothness Mapping in Metallic Workflow
_SPECGLOSSMAP	Specular/Smoothness Mapping in Specular Workflow

Using the keywords above is enough to get your scripted Material modifications working while running in the editor.

However, because Unity only checks for Materials used in your project to determine which variants to include in your build, it will not include variants that are *only* encountered via script at runtime.

This means if you enable the _PARALLAXMAP keyword for a Material in your script, but you do not have a Material used in your project matching that same feature combination, the parallax mapping will not work in your final build - even though it appears to work in the editor. This is because that variant will have been omitted from the build because it seemed to not be required.

2. You must make sure Unity includes your required shader variants in the build

To do this, you need to make sure Unity knows that you want to use that shader variant by including at least one Material of that type in your Assets. The material **must be used in a scene** or alternatively be placed in your Resources Folder - otherwise Unity will still omit it from your build, because it appeared unused.

By completing both of the above steps, you have the full ability to modify your Materials using the Standard Shader at runtime.

If you are interested in learning more about the details of shader variants, and how to write your own, read about Making multiple shader program variants here.

Writing Shaders

Leave feedback

Shaders in Unity can be written in one of three different ways:

Surface Shaders

Surface Shaders are your best option if your **Shader** needs to be affected by lights and shadows. Surface Shaders make it easy to write complex Shaders in a compact way - it's a higher level of abstraction for interaction with Unity's lighting pipeline. Most Surface Shaders automatically support both forward and deferred lighting. You write Surface Shaders in a couple of lines of **Cg/HLSL**, and a lot more code gets auto-generated from that.

Do not use Surface Shaders if your Shader is not doing anything with lights. For post-processed effects or many special-effect Shaders, Surface Shaders are a suboptimal option, since they do a bunch of lighting calculations for no good reason.

Vertex and Fragment Shaders

Vertex and Fragment Shaders are required if your Shader doesn't need to interact with lighting, or if you need some very exotic effects that the Surface Shaders can't handle. Shader programs written this way are the most flexible way to create the effect you need (even Surface Shaders are automatically converted to a bunch of Vertex and Fragment Shaders), but that comes at a price: you have to write more code and it's harder to make it interact with lighting. These Shaders are written in **Cg/HLSL** as well.

Fixed Function Shaders

Fixed Function Shaders are legacy Shader syntax for very simple effects. It is advisable to write programmable Shaders, since that allows much more flexibility. Fixed function shaders are entirely written in a language called **ShaderLab** , which is similar to Microsoft's .FX files or NVIDIA's CgFX. Internally, all Fixed Function Shaders are converted into Vertex and Fragment Shaders at shader import time.

ShaderLab

Regardless of which type you choose, the actual Shader code is always wrapped in ShaderLab, which is used to organize the Shader structure. It looks like this:

```
Shader "MyShader" {
    Properties {
        _MyTexture ("My Texture", 2D) = "white" { }
        // Place other properties like colors or vectors here as well
    }
    SubShader {
        // here goes your
        // - Surface Shader or
        // - Vertex and Fragment Shader or
        // - Fixed Function Shader
    }
    SubShader {
        // Place a simpler "fallback" version of the SubShader above
```

```
        // that can run on older graphics cards here
    }
}
```

We recommend that you start by reading about some basic concepts of the ShaderLab syntax in the ShaderLab reference and then move on to the tutorials listed below.

The tutorials include plenty of examples for the different types of Shaders. Unity's post-processing effects allows you to create many interesting effects with shaders.

Read on for an introduction to shaders, and check out the Shader reference!

Tutorial: ShaderLab & Fixed Function Shaders
Tutorial: Vertex and Fragment Shaders
Surface Shaders
Writing Vertex and Fragment Shaders

Legacy Shaders

Leave feedback

Prior to the introduction of the Physically Based Standard Shader, Unity was supplied with more than eighty built-in **shaders** which each served different purposes. These **shaders** are still included and available for use in Unity for backwards compatibility but we recommend you use the **Standard Shader** wherever possible for new projects.

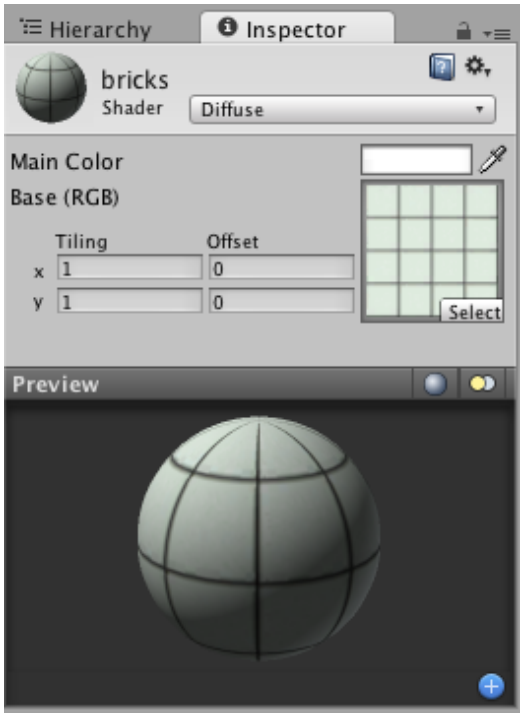
This section begins by explaining how to use the legacy built-in shaders to maximum effect. The remainder of the section details all the available shaders, grouped into related “families”.

Usage and Performance of Built-in Shaders

Leave feedback

Shaders in Unity are used through **Materials**, which essentially combine **shader** code with parameters like textures. An in-depth explanation of the **Shader**/Material relationship can be read here.

Material properties will appear in the **Inspector** when either the Material itself or a **GameObject** that uses the Material is selected. The Material Inspector looks like this:



Each Material will look a little different in the Inspector, depending on the specific shader it is using. The shader iself determines what kind of properties will be available to adjust in the Inspector. Material inspector is described in detail in Material reference page. Remember that a shader is implemented through a Material. So while the shader defines the properties that will be shown in the Inspector, each Material actually contains the adjusted data from sliders, colors, and textures. The most important thing to remember about this is that a single shader can be used in multiple Materials, but a single Material cannot use multiple shaders.

Performance Considerations

There are a number of factors that can affect the overall performance of your game. This page will talk specifically about the performance considerations for Built-in Shaders. Performance of a shader mostly depends on two things: shader itself and which Rendering Path is used by the project or specific **camera**. For performance tips when writing your own shaders, see ShaderLab Shader Performance page.

Rendering Paths and shader performance

From the **rendering** paths Unity supports, Deferred Shading and Vertex Lit paths have the most predictable performance. In Deferred shading, each object is generally drawn once, no matter what lights are affecting it. Similarly, in Vertex Lit each object is generally drawn once. So then, the performance differences in shaders mostly depend on how many textures they use and what calculations they do.

Shader Performance in Forward rendering path

In Forward rendering path, performance of a shader depends on **both** the shader itself and the lights in the **scene**. The following section explains the details. There are two basic categories of shaders from a performance perspective, **Vertex-Lit**, and **Pixel-Lit**.

Vertex-Lit shaders in **Forward rendering** path are always cheaper than Pixel-Lit shaders. These shaders work by calculating lighting based on the **mesh** vertices, using all lights at once. Because of this, no matter how many lights are shining on the object, it will only have to be drawn once.

Pixel-Lit shaders calculate final lighting at each pixel that is drawn. Because of this, the object has to be drawn once to get the ambient & main directional light, and once for each additional light that is shining on it. Thus the formula is N rendering passes, where N is the final number of pixel lights shining on the object. This increases the load on the CPU to process and send off commands to the graphics card, and on the graphics card to process the vertices and draw the **pixels**. The size of the Pixel-lit object on the screen will also affect the speed at which it is drawn. The larger the object, the slower it will be drawn.

So pixel lit shaders come at performance cost, but that cost allows for some spectacular effects: shadows, normal-mapping, good looking specular highlights and light cookies, just to name a few.

Remember that lights can be forced into a pixel ("important") or vertex/SH ("not important") mode. Any vertex lights shining on a Pixel-Lit shader will be calculated based on the object's vertices or whole object, and will not add to the rendering cost or visual effects that are associated with pixel lights.

General shader performance

Out of Built-in Shaders, they come roughly in this order of increasing complexity:

Unlit. This is just a texture, not affected by any lighting.
VertexLit.
Diffuse.
Normal mapped. This is a bit more expensive than Diffuse: it adds one more texture (normal map), and a couple of shader instructions.
Specular. This adds specular highlight calculation.
Normal Mapped Specular. Again, this is a bit more expensive than Specular.
Parallax Normal mapped. This adds parallax normal-mapping calculation.
Parallax Normal Mapped Specular. This adds both parallax normal-mapping and specular highlight calculation.

Mobile simplified shaders

Additionally, Unity has several simplified shaders targeted at mobile platforms, under "Mobile" category. These shaders work on other platforms as well, so if you can live with their simplifications (e.g. approximate specular, no per-material color support etc.), try using them!

To see the specific simplifications that have been made for each shader, look at the `.shader` files from the "built-in shaders" package and the information is listed at the top of the file in some comments.

Some examples of changes that are common across the Mobile shaders are:

There is no material color or main color for tinting the shader.
For the shaders taking a normal map, the tiling and offset from the base texture is used.
The particle shaders do not support `AlphaTest` or `ColorMask`.

Limited feature and lighting support - e.g. some shaders only support one directional light.

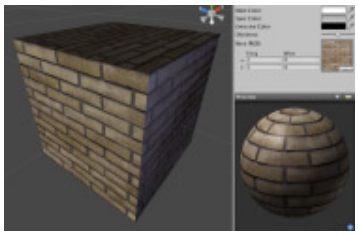
Normal Shader Family

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces these **shaders**.

These shaders are the basic shaders in Unity. They are not specialized in any way and should be suitable for most opaque objects. They are not suitable if you want your object to be transparent, emitting light **etc**.

Vertex Lit

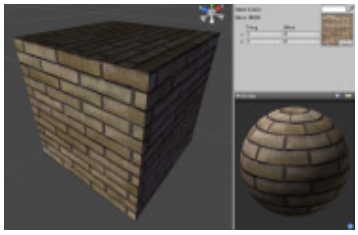


shader-NormalVertexLit

Assets needed:

One **Base** texture, no alpha channel required

Diffuse

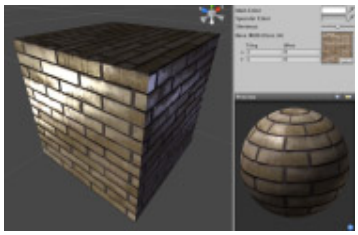


shader-NormalDiffuse

Assets needed:

One **Base** texture, no alpha channel required

Specular

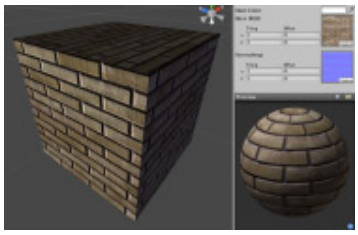


shader-NormalSpecular

Assets needed:

One **Base** texture with alpha channel for Specular Map

Normal mapped

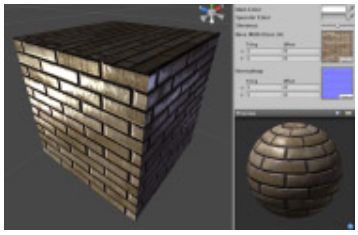


shader-NormalBumpedDiffuse

Assets needed:

One **Base** texture, no alpha channel required
One **Normal map**

Normal mapped Specular

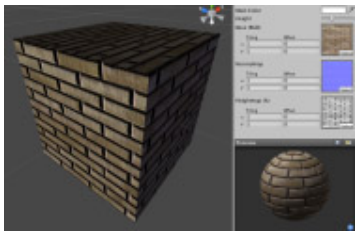


shader-NormalBumpedSpecular

Assets needed:

One **Base** texture with alpha channel for Specular Map
One **Normal map**

Parallax

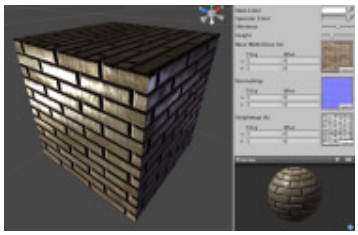


shader-NormalParallaxDiffuse

Assets needed:

One **Base** texture, no alpha channel required
One **Normal map**
One **Height** texture with Parallax Depth in the alpha channel

Parallax Specular

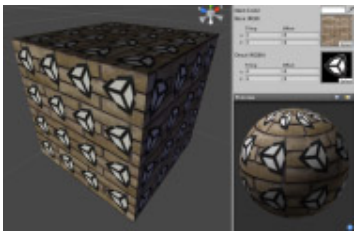


shader-NormalParallaxSpecular

Assets needed:

One **Base** texture with alpha channel for Specular Map
One **Normal map**
One **Height** texture with Parallax Depth in the alpha channel

Decal

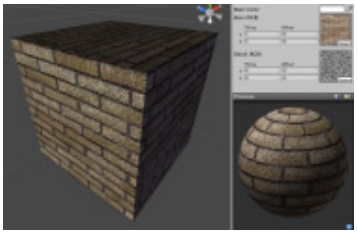


shader-NormalDecal

Assets needed:

One **Base** texture, no alpha channel required
One **Decal** texture with alpha channel for Decal transparency

Diffuse Detail



shader-NormalDiffuseDetail

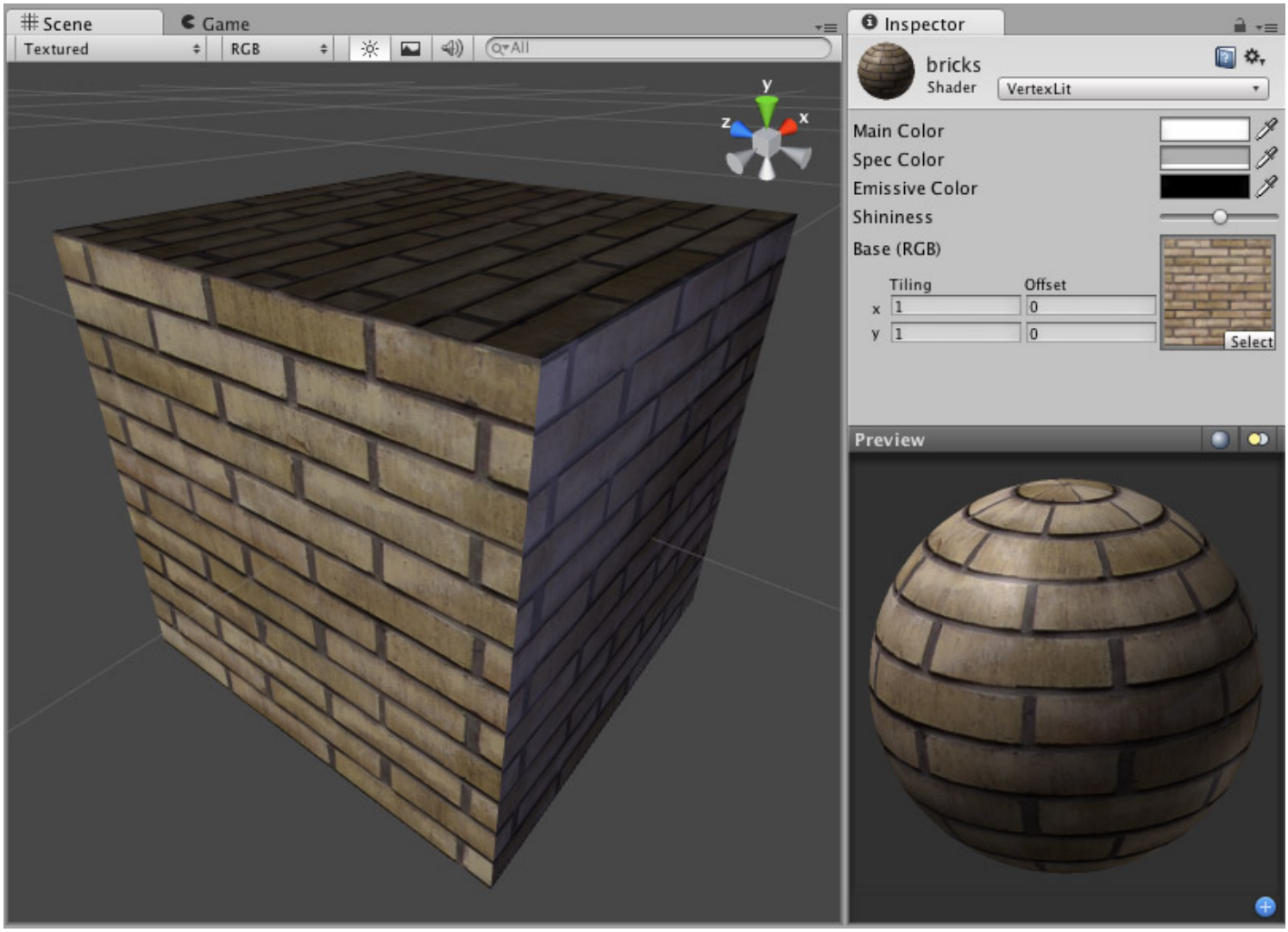
Assets needed:

One **Base** texture, no alpha channel required
One **Detail** grayscale texture; with 50% gray being neutral color

Vertex-Lit

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Vertex-Lit Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based **rendering** effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

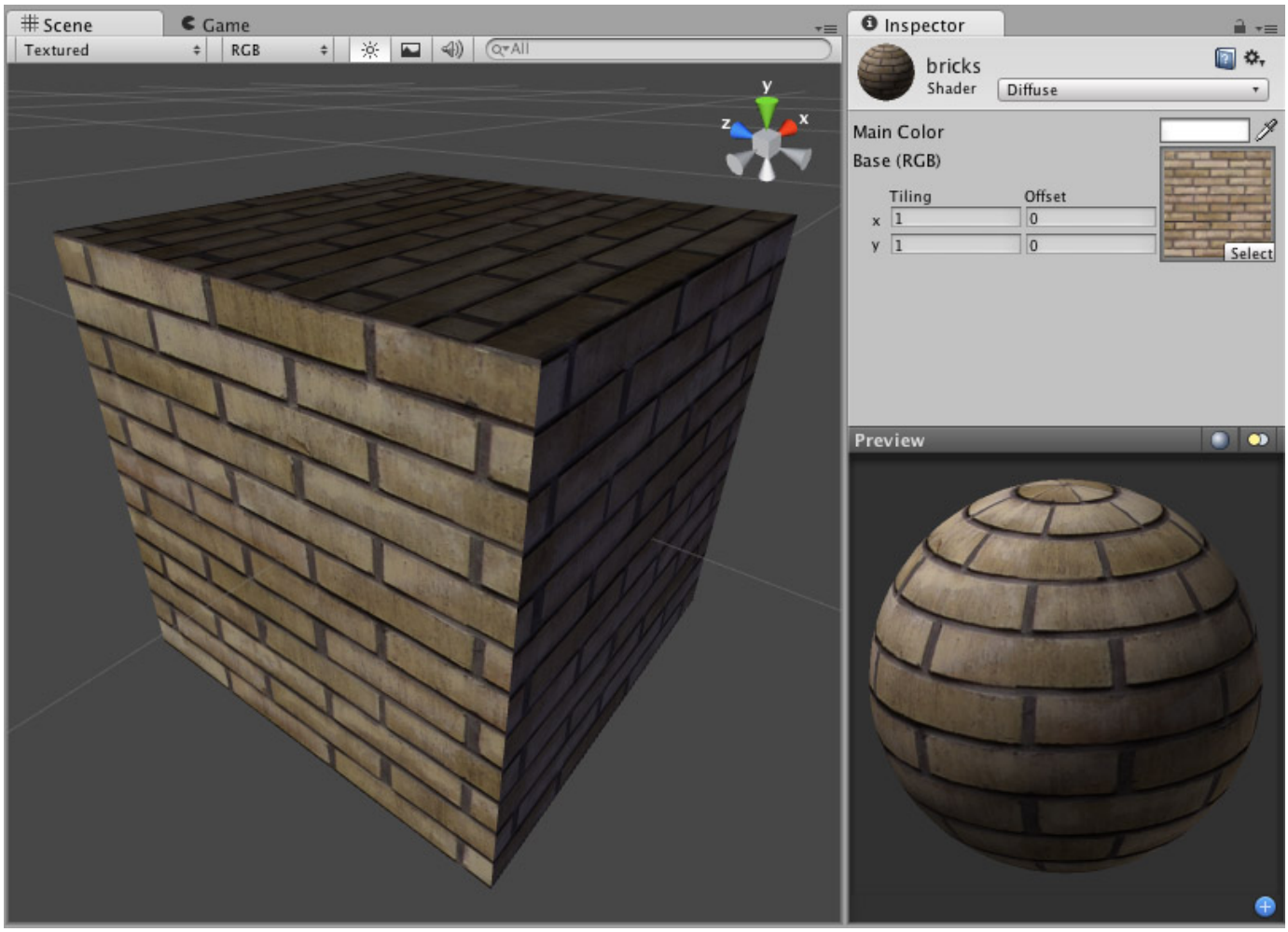
Performance

Generally, this shader is very cheap to render. For more details, please view the Shader Peformance page.

Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the **camera** moves or rotates around.

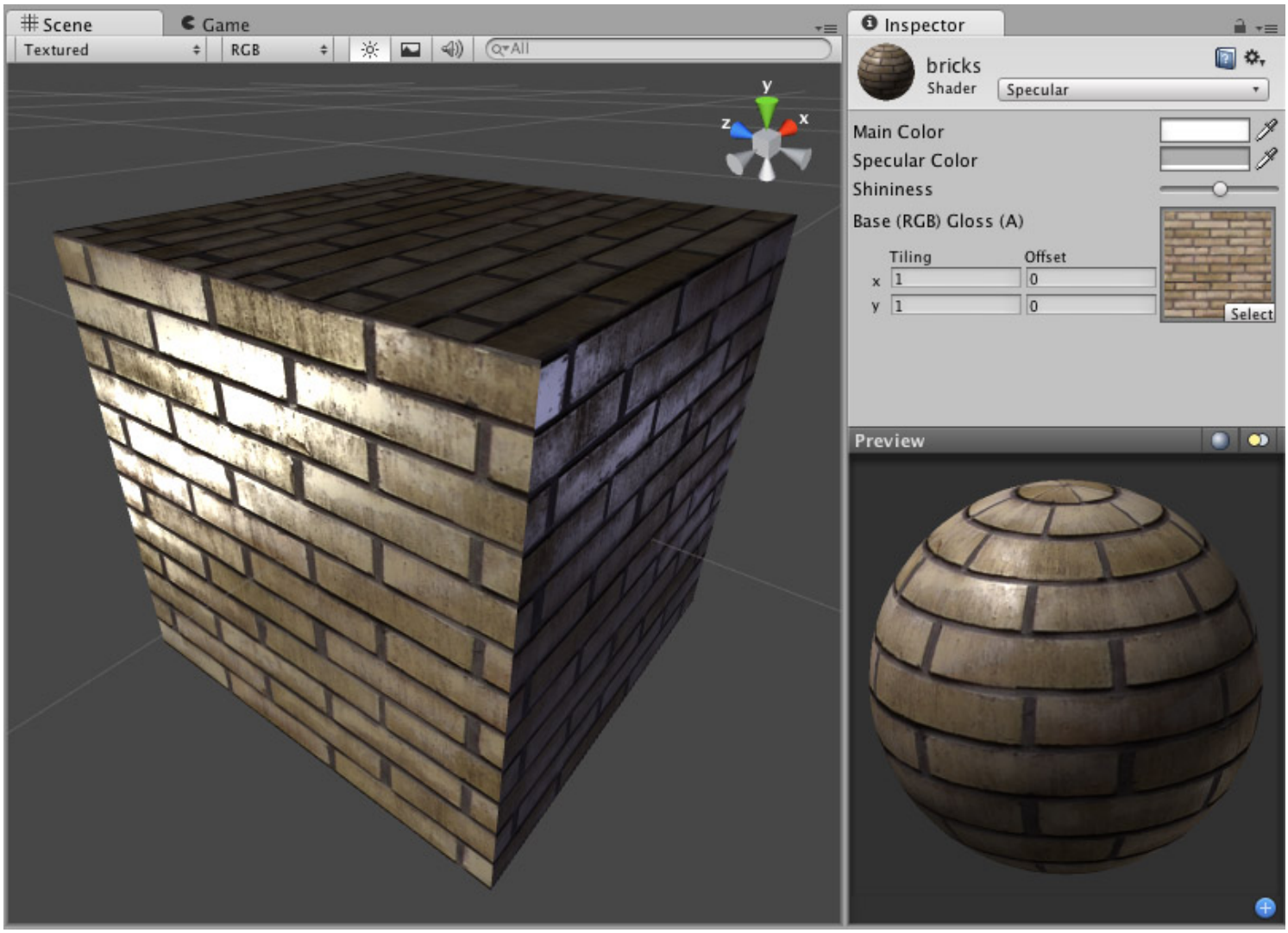
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called “gloss map”), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

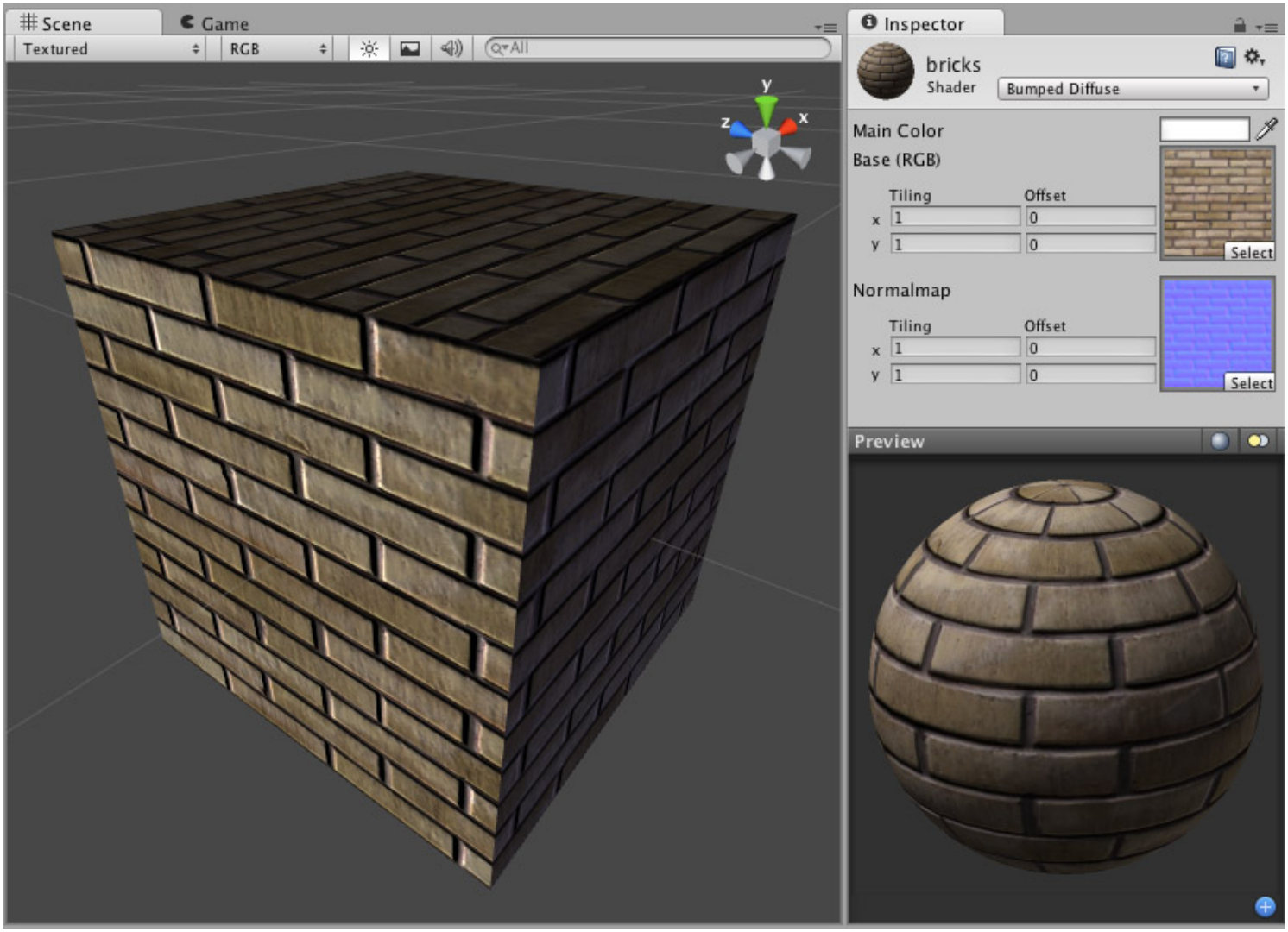
Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Bumped Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the **mesh**'s geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that “follows the surface” of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other “object space” type Normal Maps, but have some advantages:

It’s possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It’s possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the camera moves or rotates around.

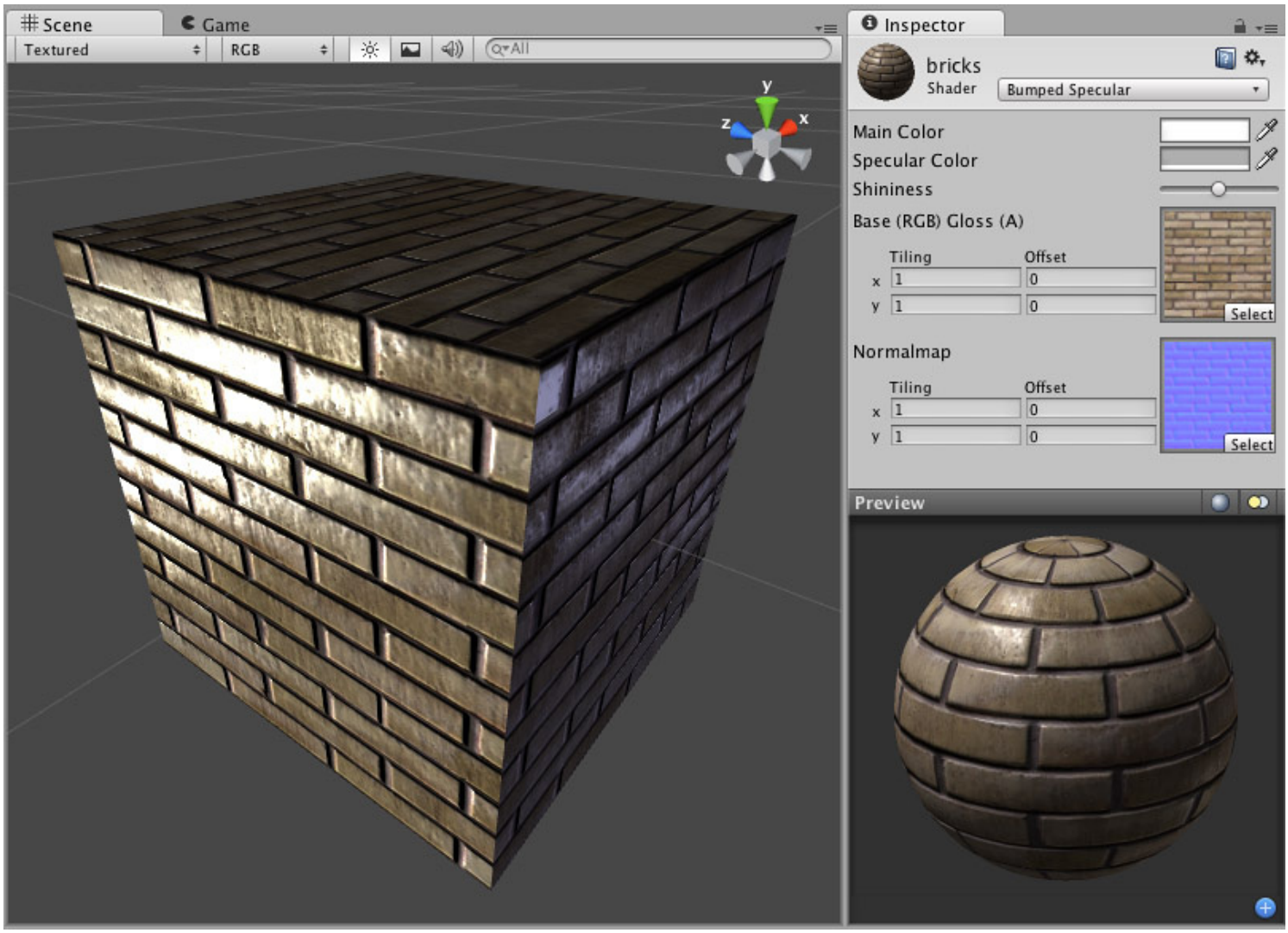
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Bumped Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the **mesh**'s geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that “follows the surface” of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other “object space” type Normal Maps, but have some advantages:

It’s possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It’s possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called “gloss map”), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

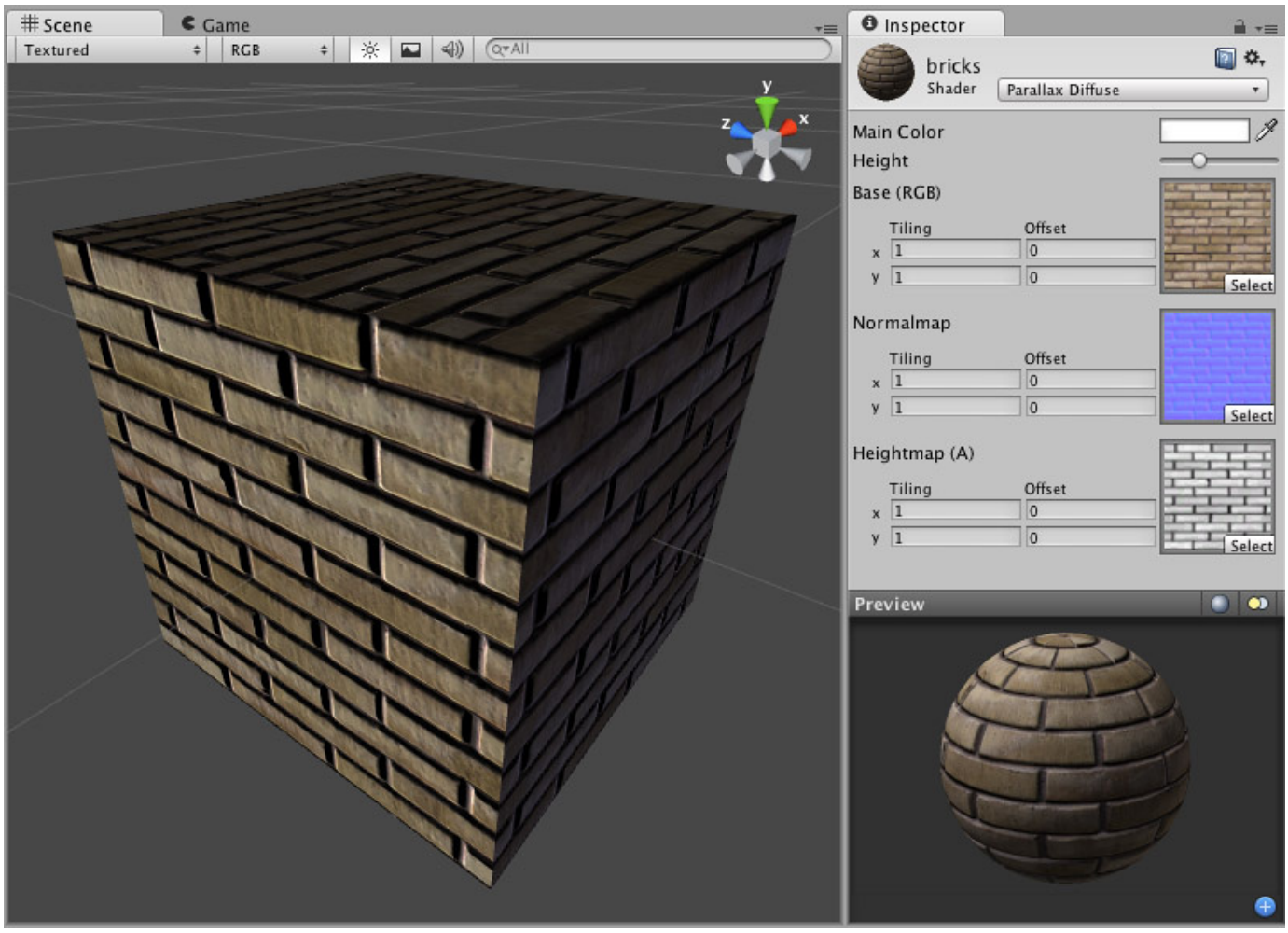
Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Parallax Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Parallax Normal mapped Properties

Parallax Normal mapped is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the **camera** moves or rotates around.

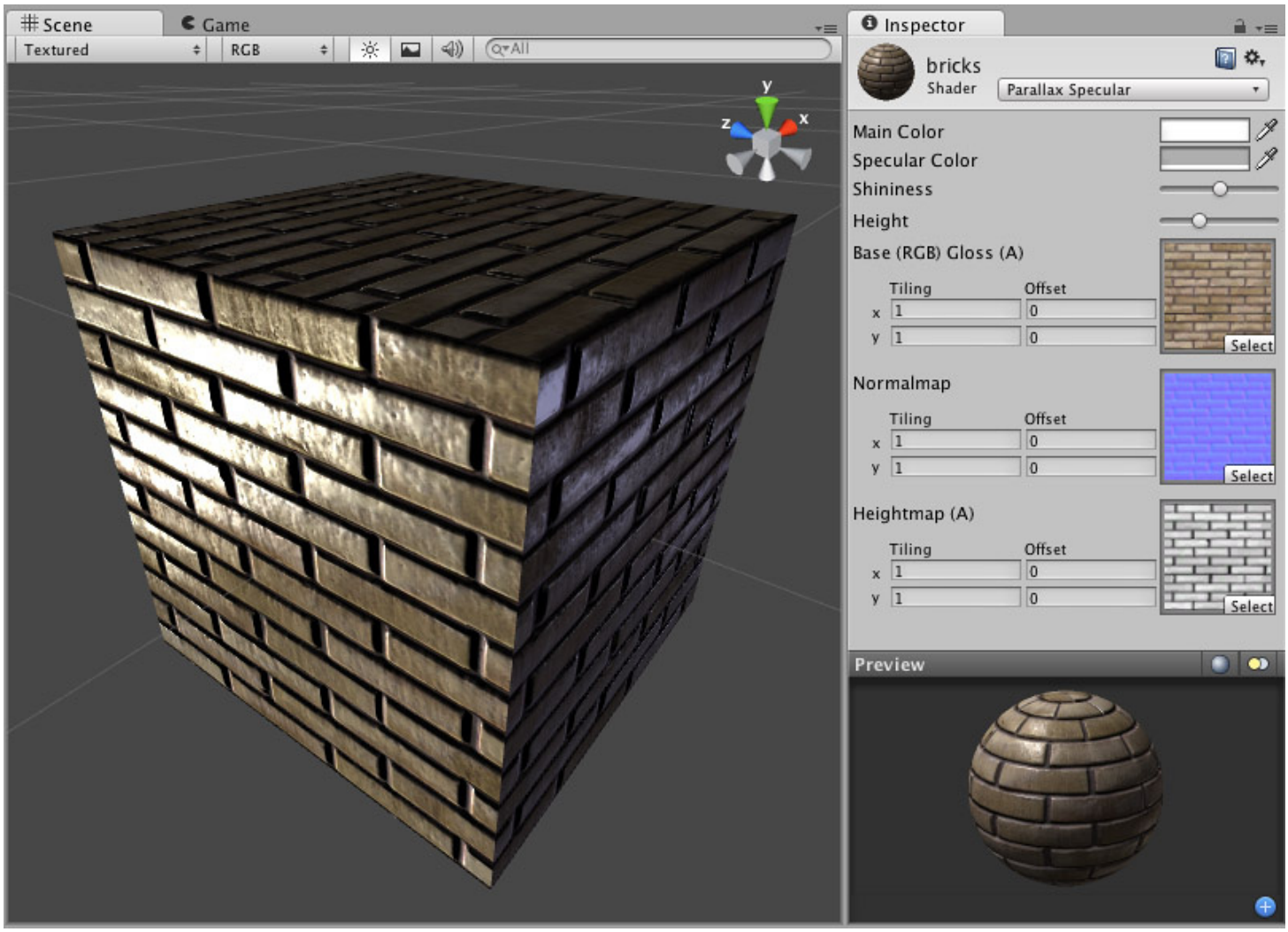
Performance

Generally, this shader is on the more expensive **rendering** side. For more details, please view the Shader Peformance page.

Parallax Bumped Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Parallax Normal mapped Properties

Parallax Normal mapped is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas

will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

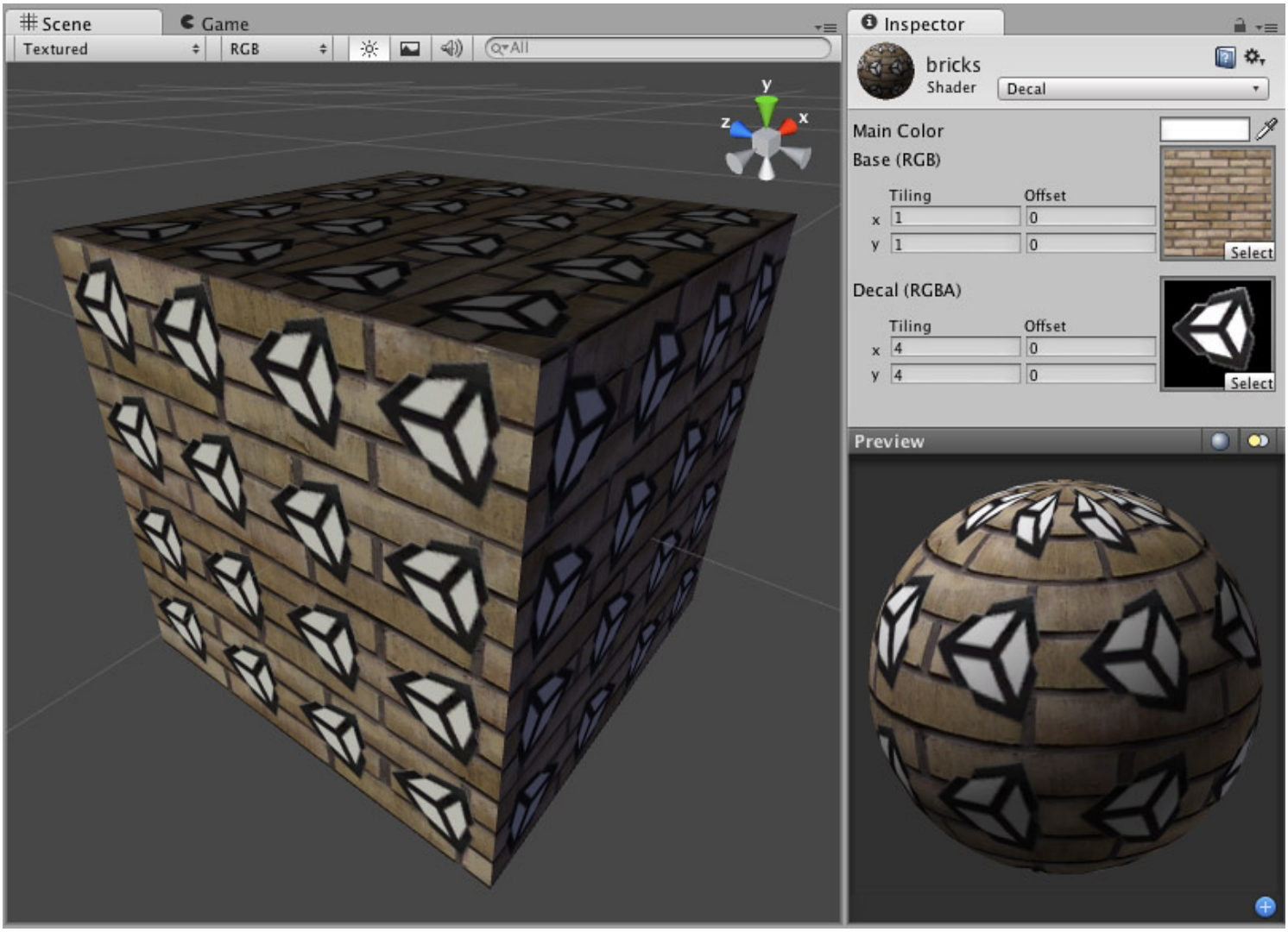
Performance

Generally, this shader is on the more expensive **rendering** side. For more details, please view the Shader Peformance page.

Decal

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Decal Properties

This shader is a variation of the VertexLit shader. All lights that shine on it will be rendered as vertex lights by this shader. In addition to the main texture, this shader makes use of a second texture for additional details. The second “Decal” texture uses an alpha channel to determine visible areas of the main texture. The decal texture should be supplemental to the main texture. For example, if you have a brick wall, you can tile the brick texture as the main texture, and use the decal texture with alpha channel to draw graffiti at different places on the wall.

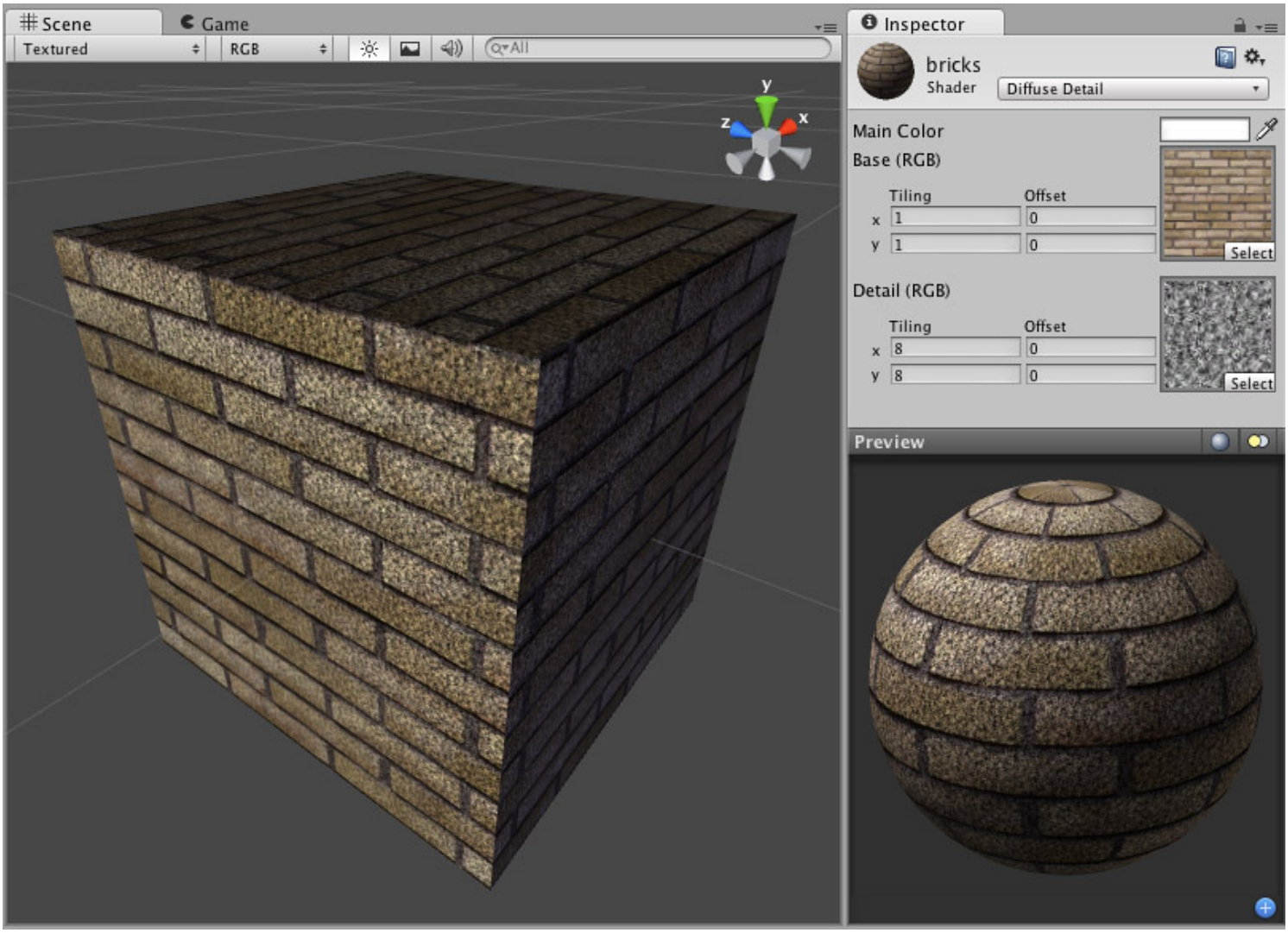
Performance

This shader is approximately equivalent to the VertexLit shader. It is marginally more expensive due to the second decal texture, but will not have a noticeable impact.

Diffuse Detail

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Diffuse Detail Properties

This shader is a version of the regular **Diffuse shader** with additional data. It allows you to define a second "Detail" texture that will gradually appear as the **camera** gets closer to it. It can be used on **terrain**, for example. You can use a base low-resolution texture and stretch it over the entire **terrain**. When the **camera** gets close the low-resolution texture will get blurry, and we don't want that. To avoid this effect, create a generic Detail texture that will be tiled over the **terrain**. This way, when the **camera** gets close, the additional details appear and the blurry effect is avoided.

The Detail texture is put "on top" of the base texture. Darker colors in the detail texture will darken the main texture and lighter colors will brighten it. Detail texture are usually gray-ish.

Performance

This shader is pixel-lit, and approximately equivalent to the Diffuse shader. It is marginally more expensive due to additional texture.

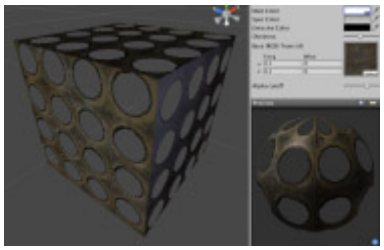
Transparent Shader Family

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces these **shaders**.

The Transparent shaders are used for fully- or semi-transparent objects. Using the alpha channel of the **Base** texture, you can determine areas of the object which can be more or less transparent than others. This can create a great effect for glass, HUD interfaces, or sci-fi effects.

Transparent Vertex-Lit



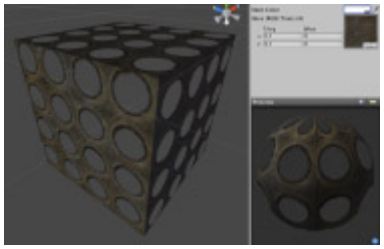
shader-TransVertexLit

Assets needed:

One **Base** texture with alpha channel for Transparency Map

» More details

Transparent Diffuse



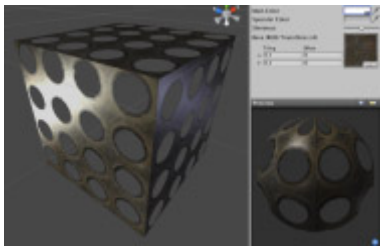
shader-TransDiffuse

Assets needed:

One **Base** texture with alpha channel for Transparency Map

» More details

Transparent Specular



shader-TransSpecular

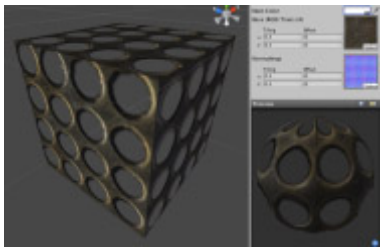
Assets needed:

One **Base** texture with alpha channel for combined Transparency Map/Specular Map

Note: One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

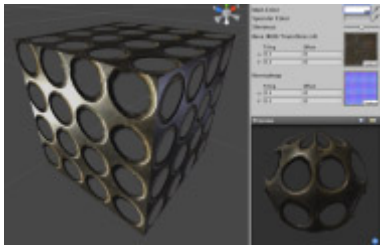
Transparent Normal mapped



shader-TransBumpedDiffuse
Assets needed:

One **Base** texture with alpha channel for Transparency Map
One **Normal map** normal map, no alpha channel required
» More details

Transparent Normal mapped Specular

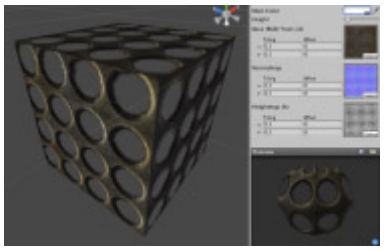


shader-TransBumpedSpecular
Assets needed:

One **Base** texture with alpha channel for combined Transparency Map/Specular Map
One **Normal map** normal map, no alpha channel required
Note: One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

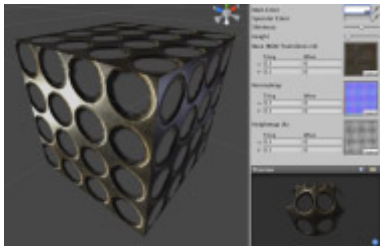
Transparent Parallax



shader-TransParallaxDiffuse
Assets needed:

One **Base** texture with alpha channel for Transparency Map
One **Normal map** normal map with alpha channel for Parallax Depth
» More details

Transparent Parallax Specular



shader-TransParallaxSpecular
Assets needed:

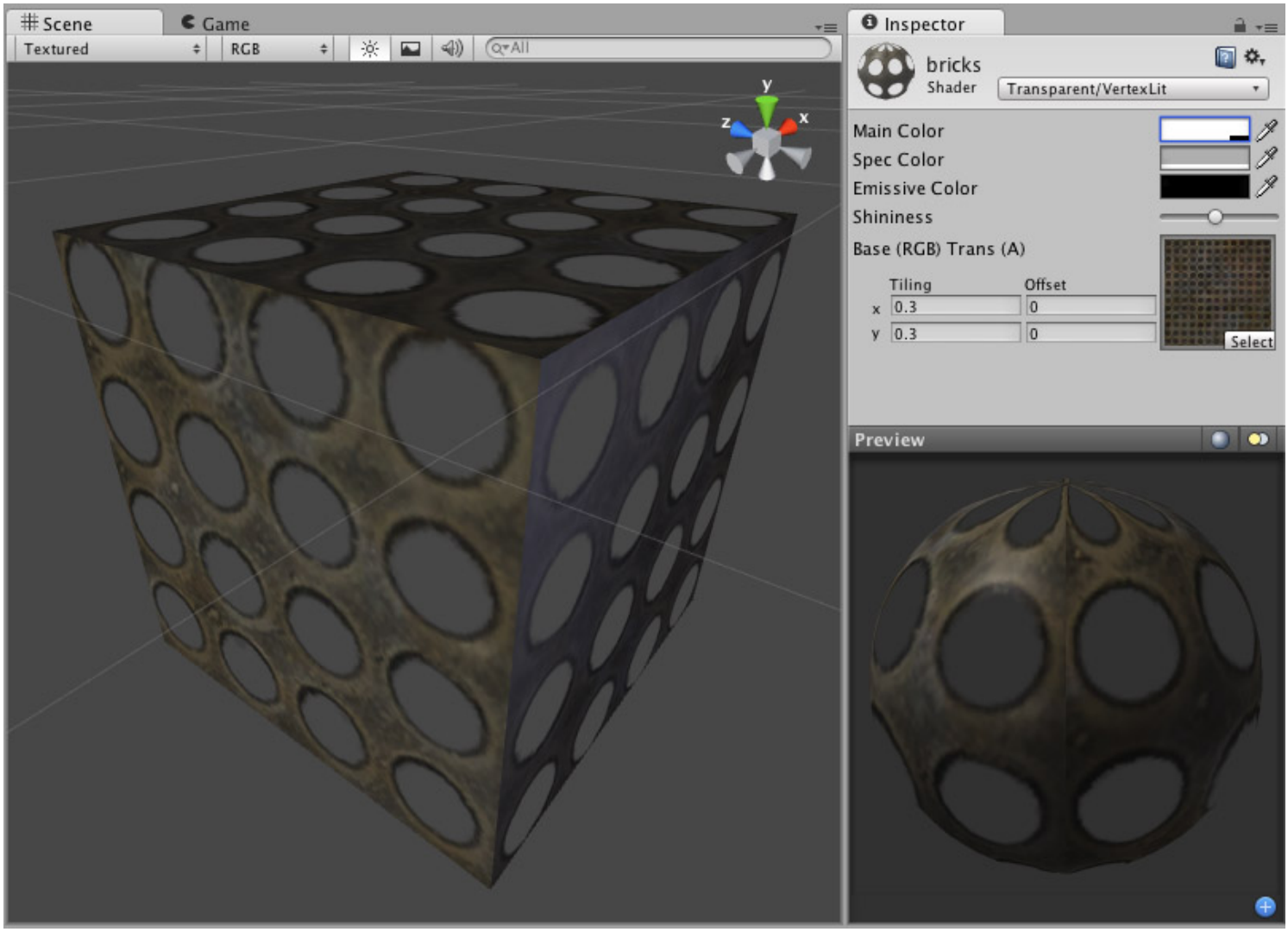
One **Base** texture with alpha channel for combined Transparency Map/Specular Map
One **Normal map** normal map with alpha channel for Parallax Depth
Note: One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

Transparent Vertex-Lit

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Transparent Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader can make **mesh** geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

Vertex-Lit Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based **rendering** effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

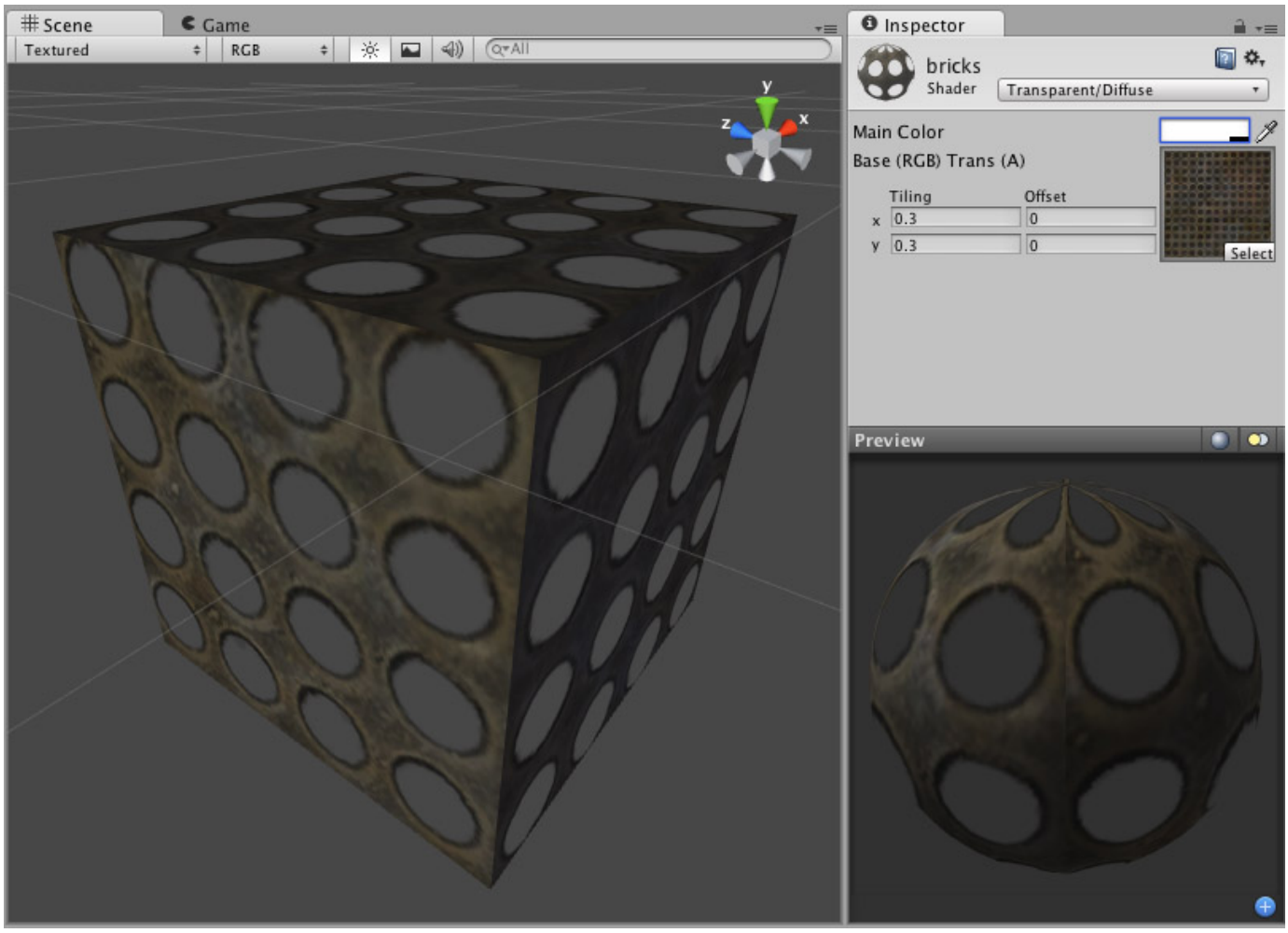
Performance

Generally, this shader is very cheap to render. For more details, please view the Shader Peformance page.

Transparent Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Transparent Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader can make **mesh** geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the **camera** moves or rotates around.

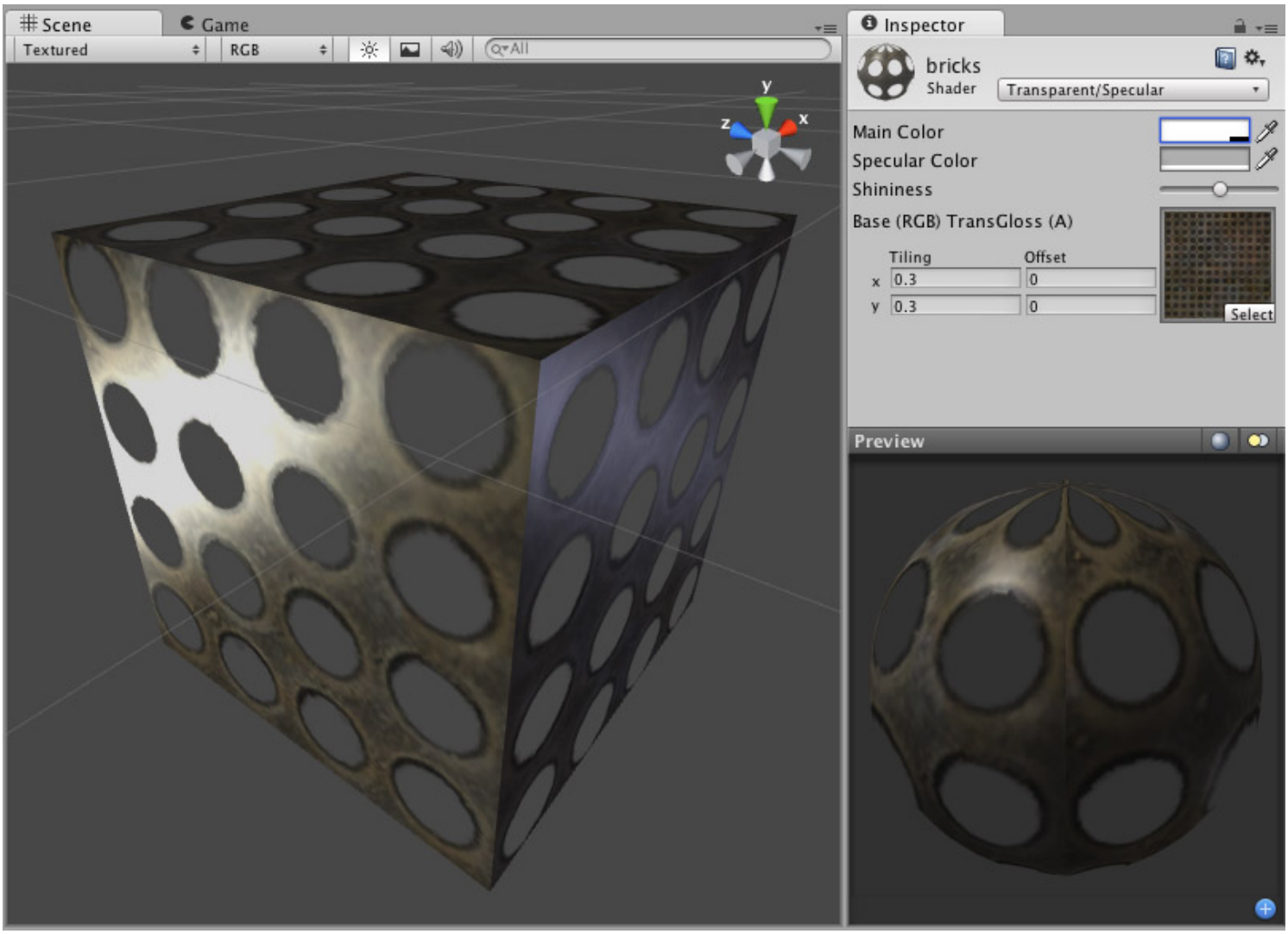
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Transparent Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

Transparent Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader can make **mesh** geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called “gloss map”), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

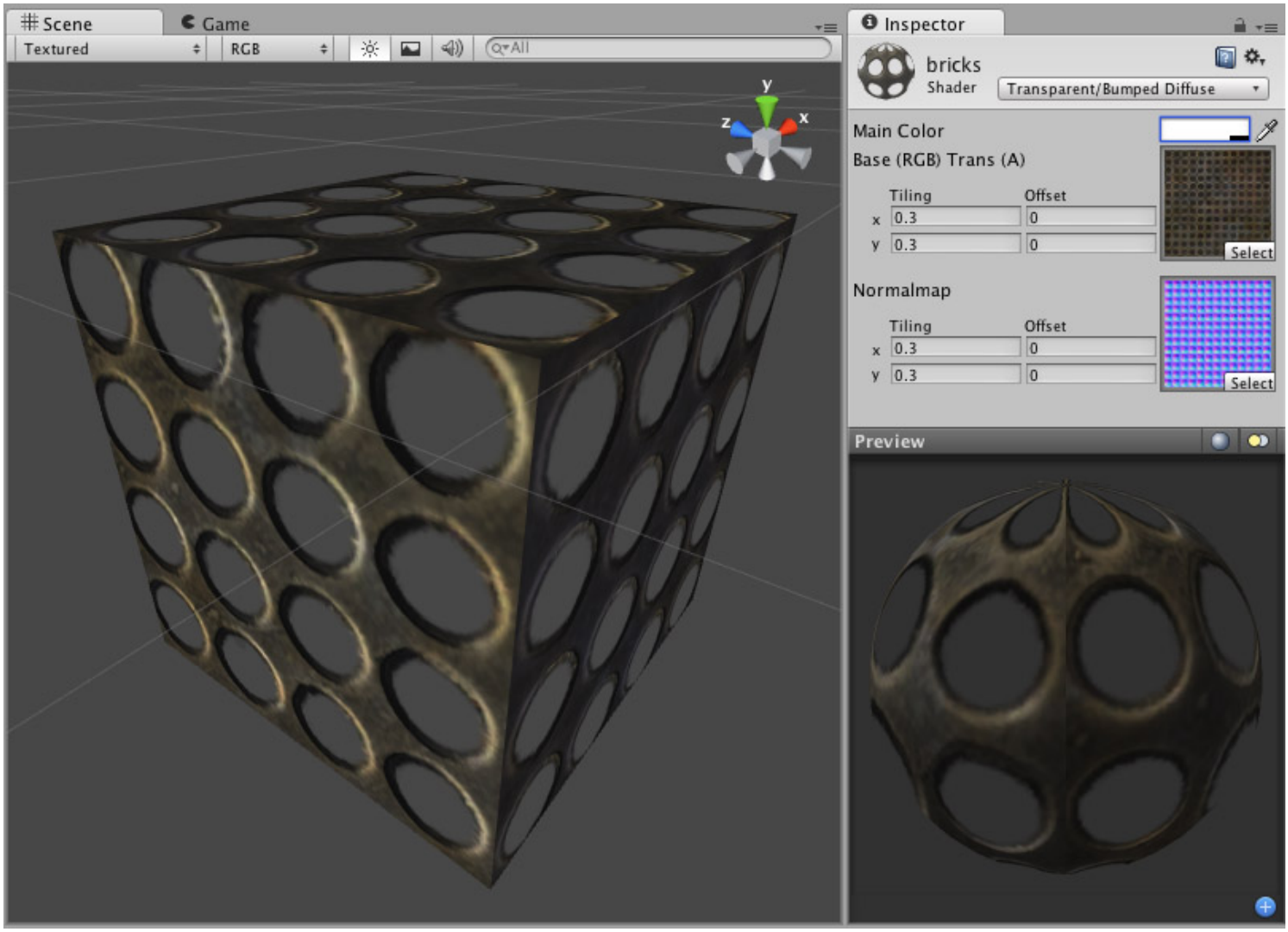
Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Transparent Bumped Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Transparent Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader can make **mesh** geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the camera moves or rotates around.

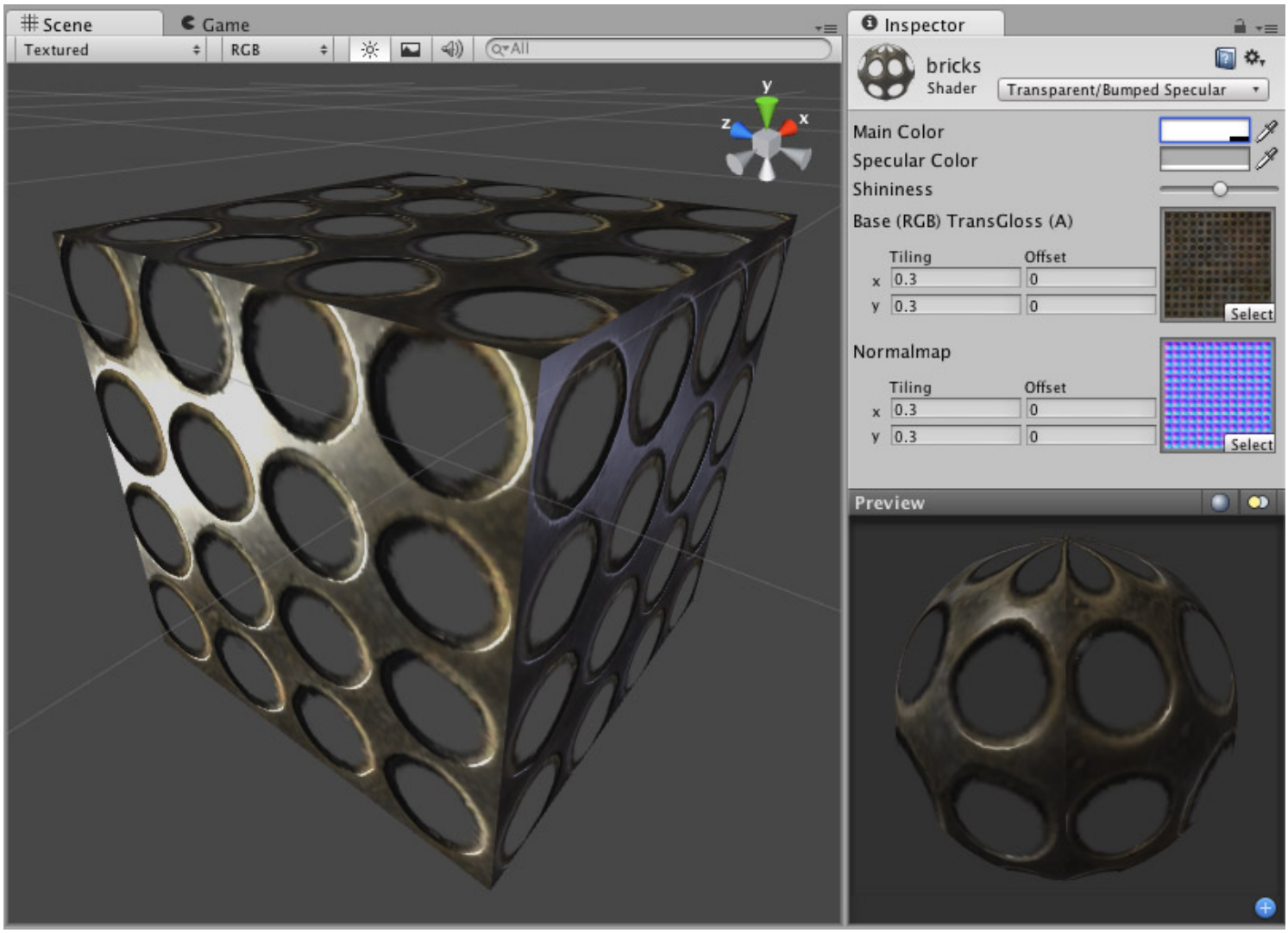
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Transparent Bumped Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

Transparent Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader can make **mesh** geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the mesh's geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

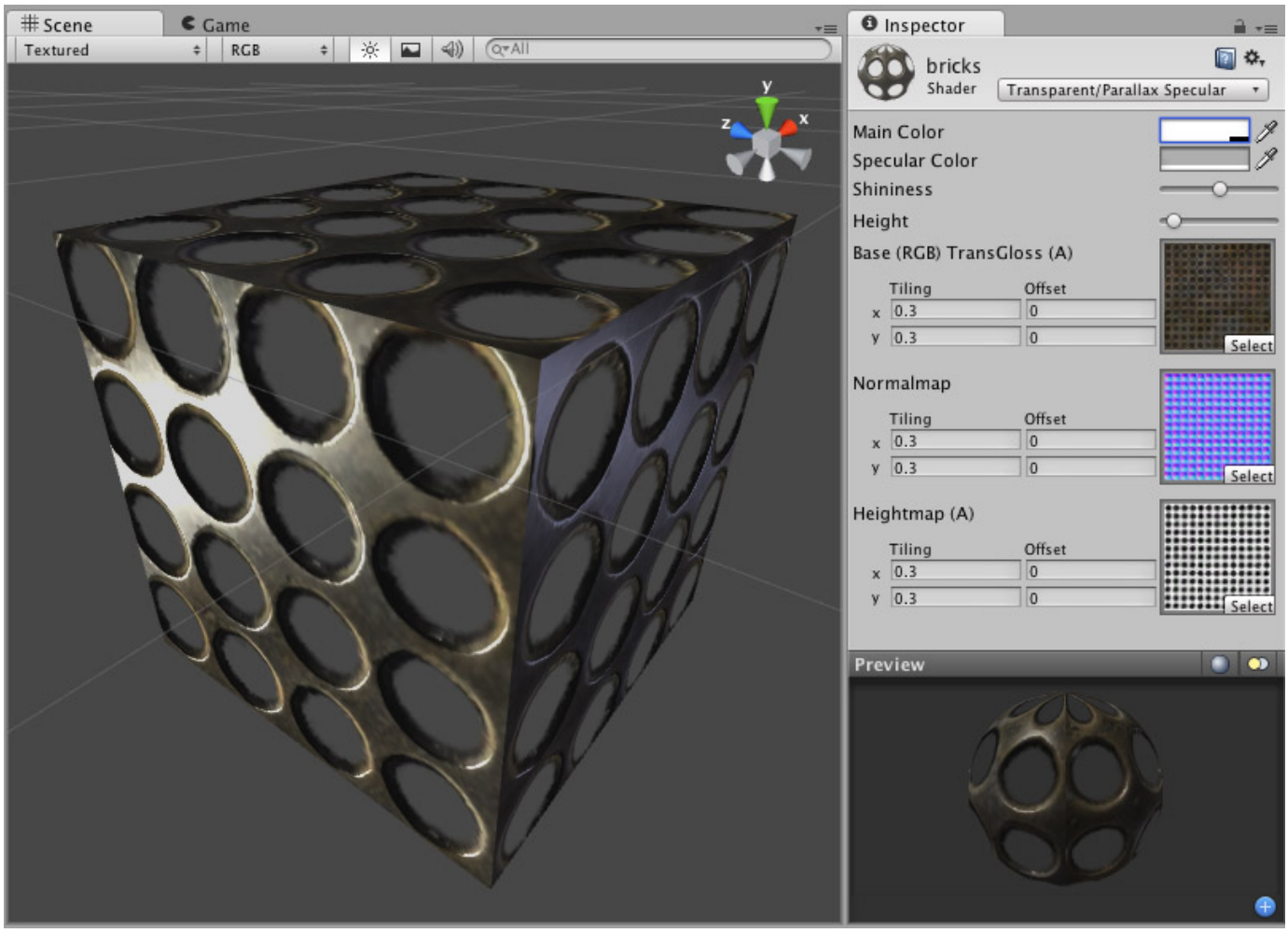
Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Transparent Parallax Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Transparent Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader can make **mesh** geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

Parallax Normal mapped Properties

Parallax Normal mapped is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an

odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the **camera** moves or rotates around.

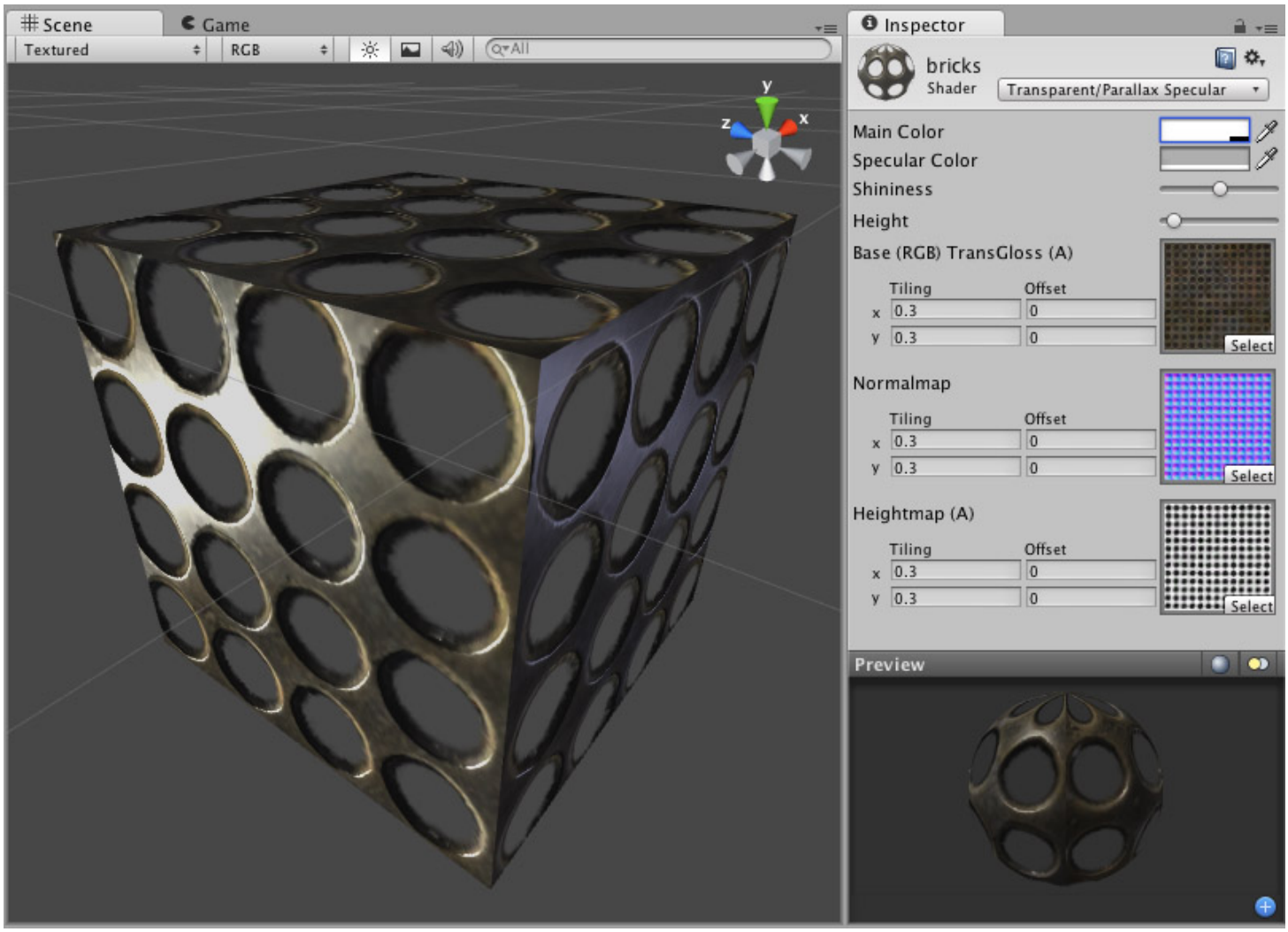
Performance

Generally, this shader is on the more expensive **rendering** side. For more details, please view the Shader Peformance page.

Transparent Parallax Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

Transparent Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader can make **mesh** geometry partially or fully transparent by reading the alpha channel of the main texture. In the alpha, 0 (black) is completely transparent while 255 (white) is completely opaque. If your main texture does not have an alpha channel, the object will appear completely opaque.

Using transparent objects in your game can be tricky, as there are traditional graphical programming problems that can present sorting issues in your game. For example, if you see odd results when looking through two windows at once, you're experiencing the classical problem with using transparency. The general rule is to be aware that there are some cases in which one transparent object may be drawn in front of another in an unusual way, especially if the objects are intersecting, enclose each other or are of very different sizes. For this reason, you should use transparent objects if you need them, and try not to let them become excessive. You should also make your designer(s) aware that such sorting problems can occur, and have them prepare to change some design to work around these issues.

Parallax Normal mapped Properties

Parallax Normal mapped is the same as regular **Normal mapped**, but with a better simulation of “depth”. The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called “gloss map”), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

Performance

Generally, this shader is on the more expensive **rendering** side. For more details, please view the Shader Peformance page.

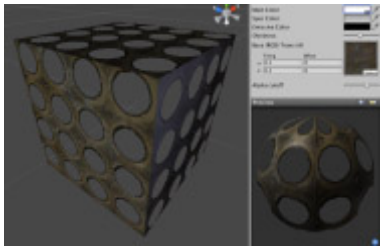
Transparent Cutout Shader Family

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces these **shaders**.

The Transparent Cutout shaders are used for objects that have fully opaque and fully transparent parts (no partial transparency). Things like chain fences, trees, grass, **etc**.

Transparent Cutout Vertex-Lit



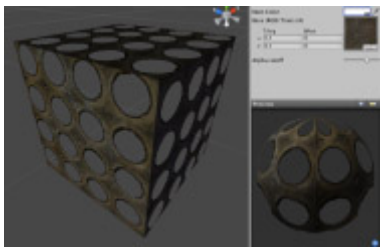
shader-TransCutVertexLit

Assets needed:

One **Base** texture with alpha channel for Transparency Map

» More details

Transparent Cutout Diffuse



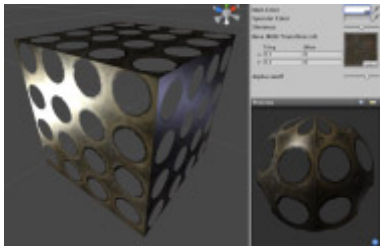
shader-TransCutDiffuse

Assets needed:

One **Base** texture with alpha channel for Transparency Map

» More details

Transparent Cutout Specular



shader-TransCutSpecular

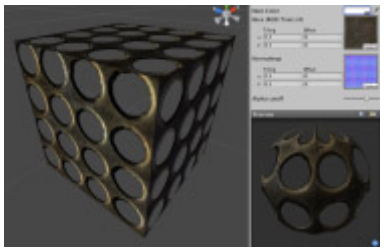
Assets needed:

One **Base** texture with alpha channel for combined Transparency Map/Specular Map

Note: One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

Transparent Cutout Bumped



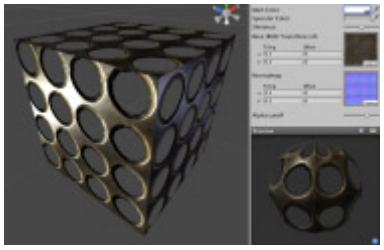
shader-TransCutBumpedDiffuse

Assets needed:

One **Base** texture with alpha channel for Transparency Map
One **Normal map** normal map, no alpha channel required

» More details

Transparent Cutout Bumped Specular



shader-TransCutBumpedSpecular

Assets needed:

One **Base** texture with alpha channel for combined Transparency Map/Specular Map
One **Normal map** normal map, no alpha channel required

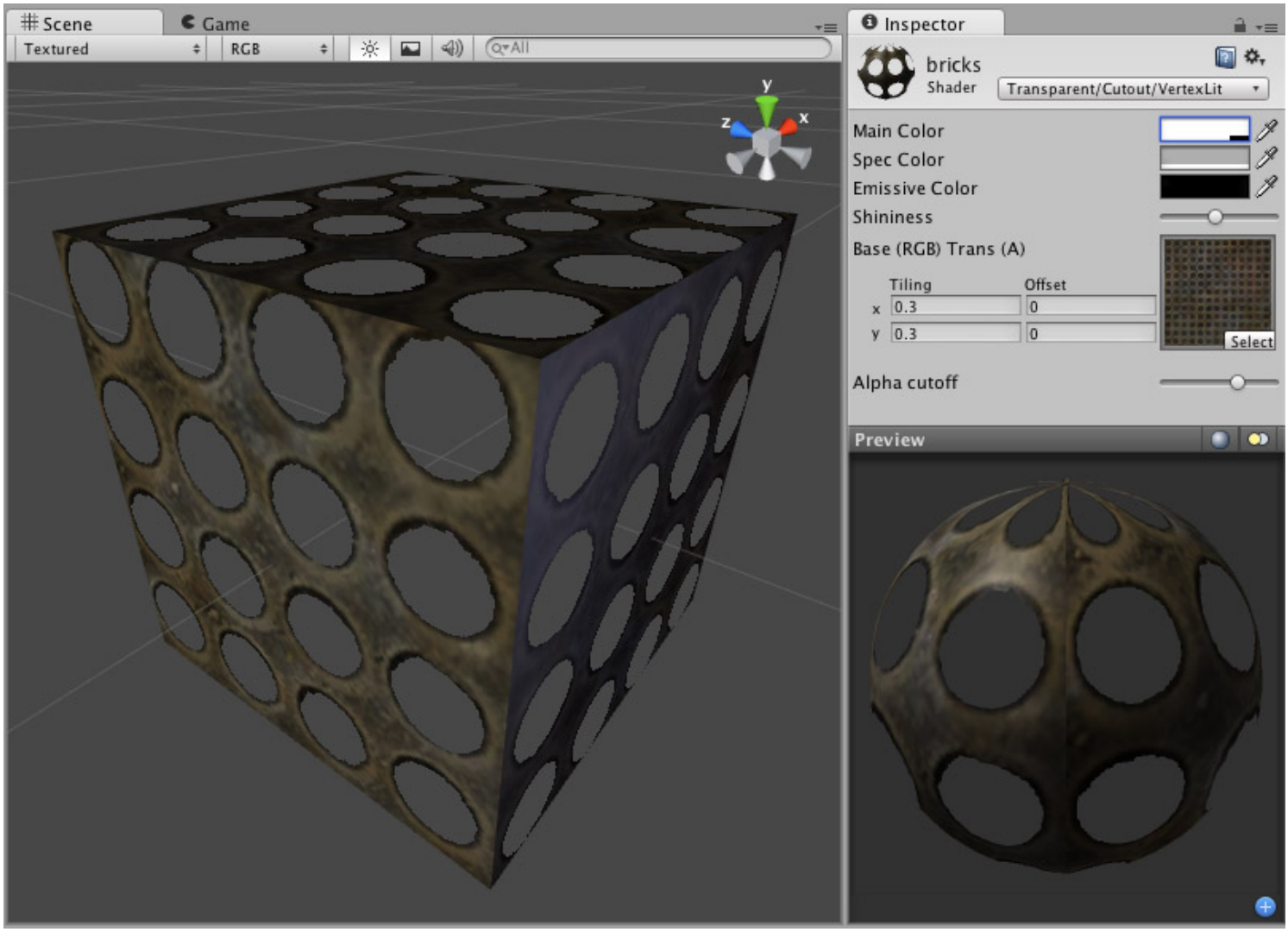
Note: One limitation of this shader is that the **Base** texture's alpha channel doubles as a Specular Map for the Specular shaders in this family.

» More details

Transparent Cutout Vertex-Lit

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Transparent Cutout Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
Objects using this shader can cast and receive shadows!
The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.
This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

Vertex-Lit Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based **rendering** effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

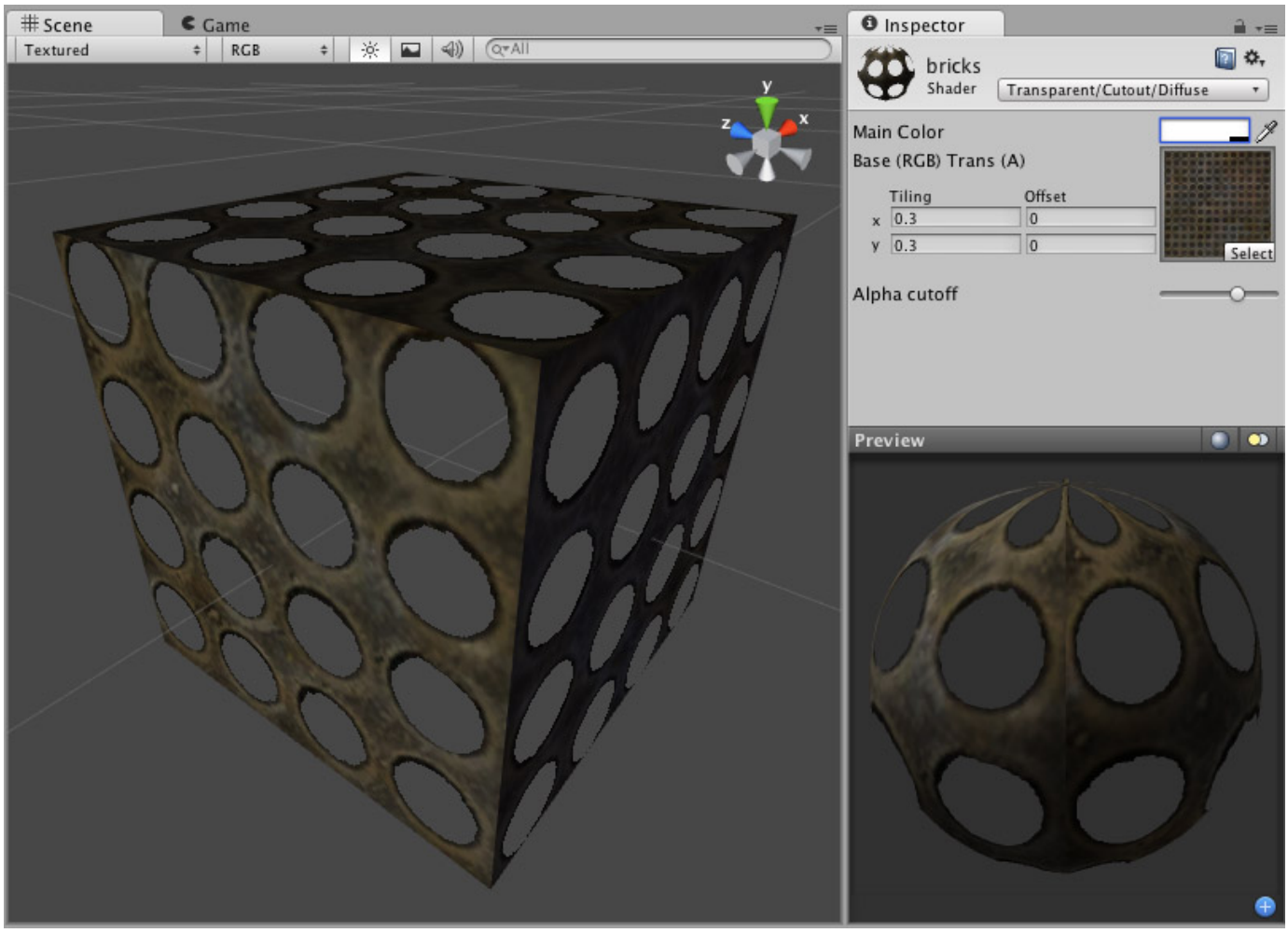
Performance

Generally, this shader is very cheap to render. For more details, please view the Shader Peformance page.

Transparent Cutout Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Transparent Cutout Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
Objects using this shader can cast and receive shadows!
The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.
This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the **camera** moves or rotates around.

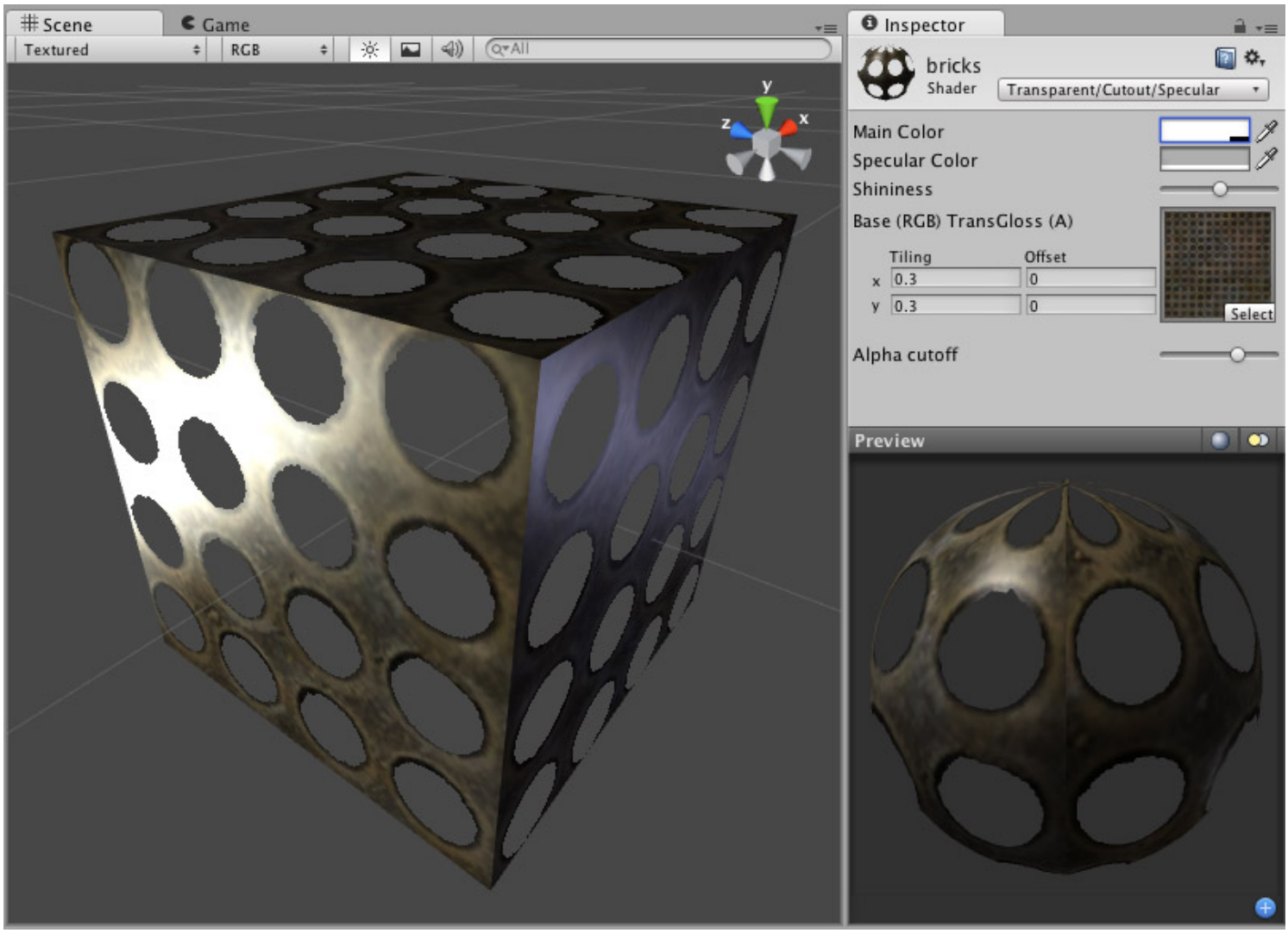
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Transparent Cutout Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

Transparent Cutout Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
Objects using this shader can cast and receive shadows!
The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.

This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called “gloss map”), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

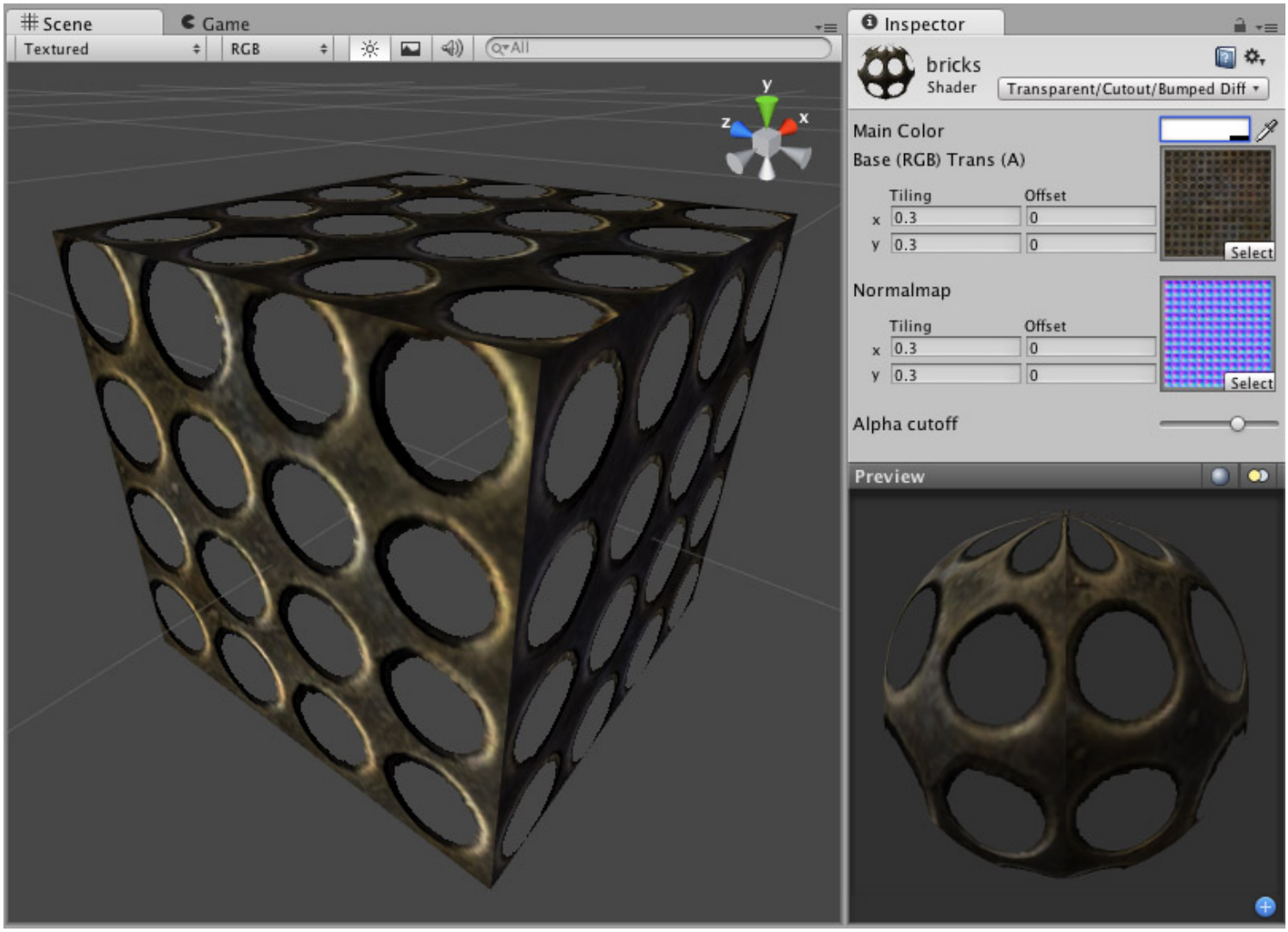
Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Transparent Cutout Bumped Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Transparent Cutout Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
Objects using this shader can cast and receive shadows!
The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.
This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the **mesh**'s geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the camera moves or rotates around.

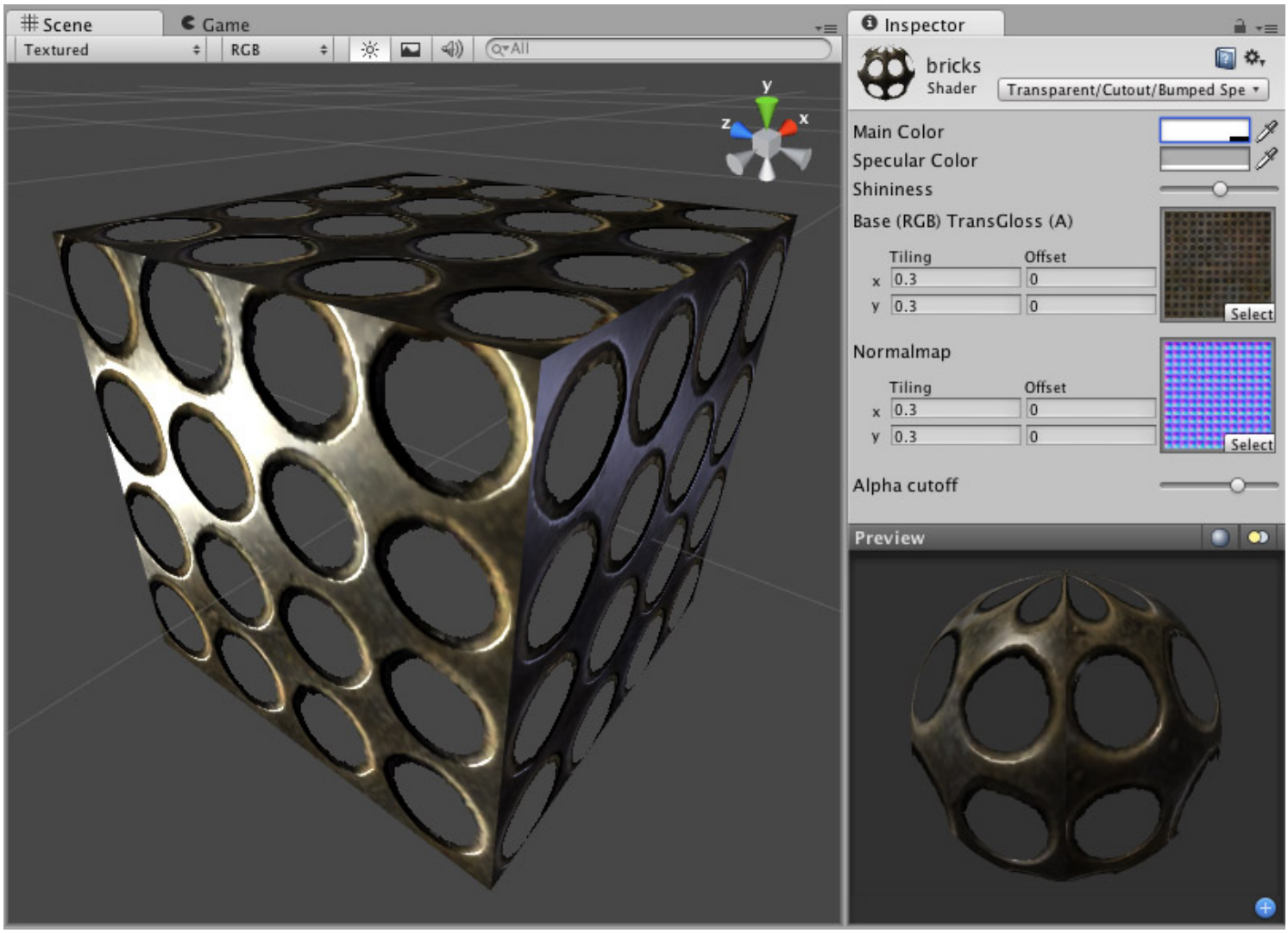
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Transparent Cutout Bumped Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



One consideration for this shader is that the Base texture's alpha channel defines both the Transparent areas as well as the Specular Map.

Transparent Cutout Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

Cutout shader is an alternative way of displaying transparent objects. Differences between Cutout and regular Transparent shaders are:

This shader cannot have partially transparent areas. Everything will be either fully opaque or fully transparent.
Objects using this shader can cast and receive shadows!
The graphical sorting problems normally associated with Transparent shaders do not occur when using this shader.

This shader uses an alpha channel contained in the **Base** Texture to determine the transparent areas. If the alpha contains a blend between transparent and opaque areas, you can manually determine the cutoff point for the which areas will be shown. You change this cutoff by adjusting the **Alpha Cutoff** slider.

Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the **mesh**'s geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Self-Illuminated Shader Family

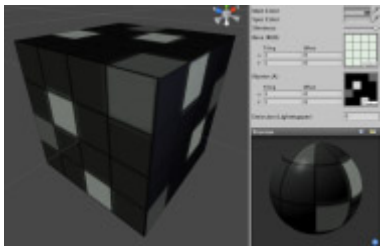
Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces these **shaders**.

The **Self-Illuminated** shaders will emit light only onto themselves based on an attached alpha channel. They do not require any Lights to shine on them to emit this light. Any vertex lights or **pixel** lights will simply add more light on top of the self-illumination.

This is mostly used for light emitting objects. For example, parts of the wall texture could be self-illuminated to simulate lights or displays. It can also be useful to light power-up objects that should always have consistent lighting throughout the game, regardless of the lights shining on it.

Self-Illuminated Vertex-Lit

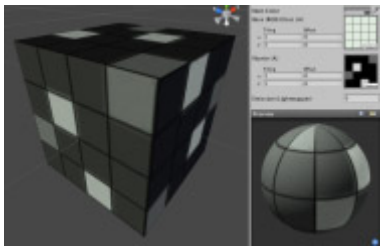


shader-SelfIllumVertexLit

Assets needed:

One **Base** texture, no alpha channel required
One **Illumination** texture with alpha channel for Illumination Map
» More details

Self-Illuminated Diffuse

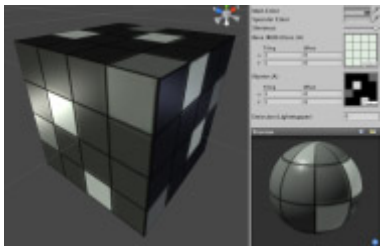


shader-SelfIllumDiffuse

Assets needed:

One **Base** texture, no alpha channel required
One **Illumination** texture with alpha channel for Illumination Map
» More details

Self-Illuminated Specular

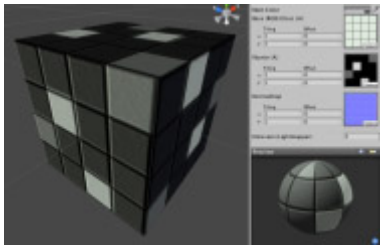


shader-SelfIllumSpecular

Assets needed:

One **Base** texture with alpha channel for Specular Map
One **Illumination** texture with alpha channel for Illumination Map
» More details

Self-Illuminated Bumped

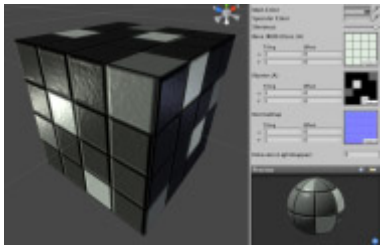


shader-SelfIllumBumpedDiffuse

Assets needed:

One **Base** texture, no alpha channel required
One **Normal map** normal map with alpha channel for Illumination
» More details

Self-Illuminated Bumped Specular

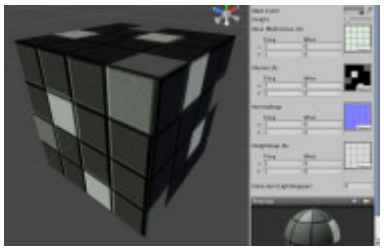


shader-SelfIllumBumpedSpecular

Assets needed:

One **Base** texture with alpha channel for Specular Map
One **Normal map** normal map with alpha channel for Illumination Map
» More details

Self-Illuminated Parallax



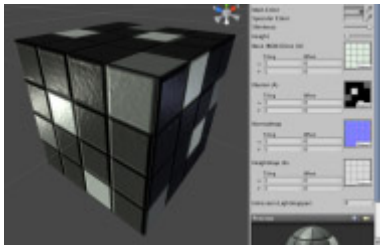
shader-SelfIllumParallaxDiffuse

Assets needed:

One **Base** texture, no alpha channel required
One **Normal map** normal map with alpha channel for Illumination Map & Parallax Depth combined
Note: One consideration of this shader is that the **Bumpmap** texture's alpha channel doubles as a Illumination and the Parallax Depth.

» More details

Self-Illuminated Parallax Specular



shader-SelfIllumParallaxSpecular

Assets needed:

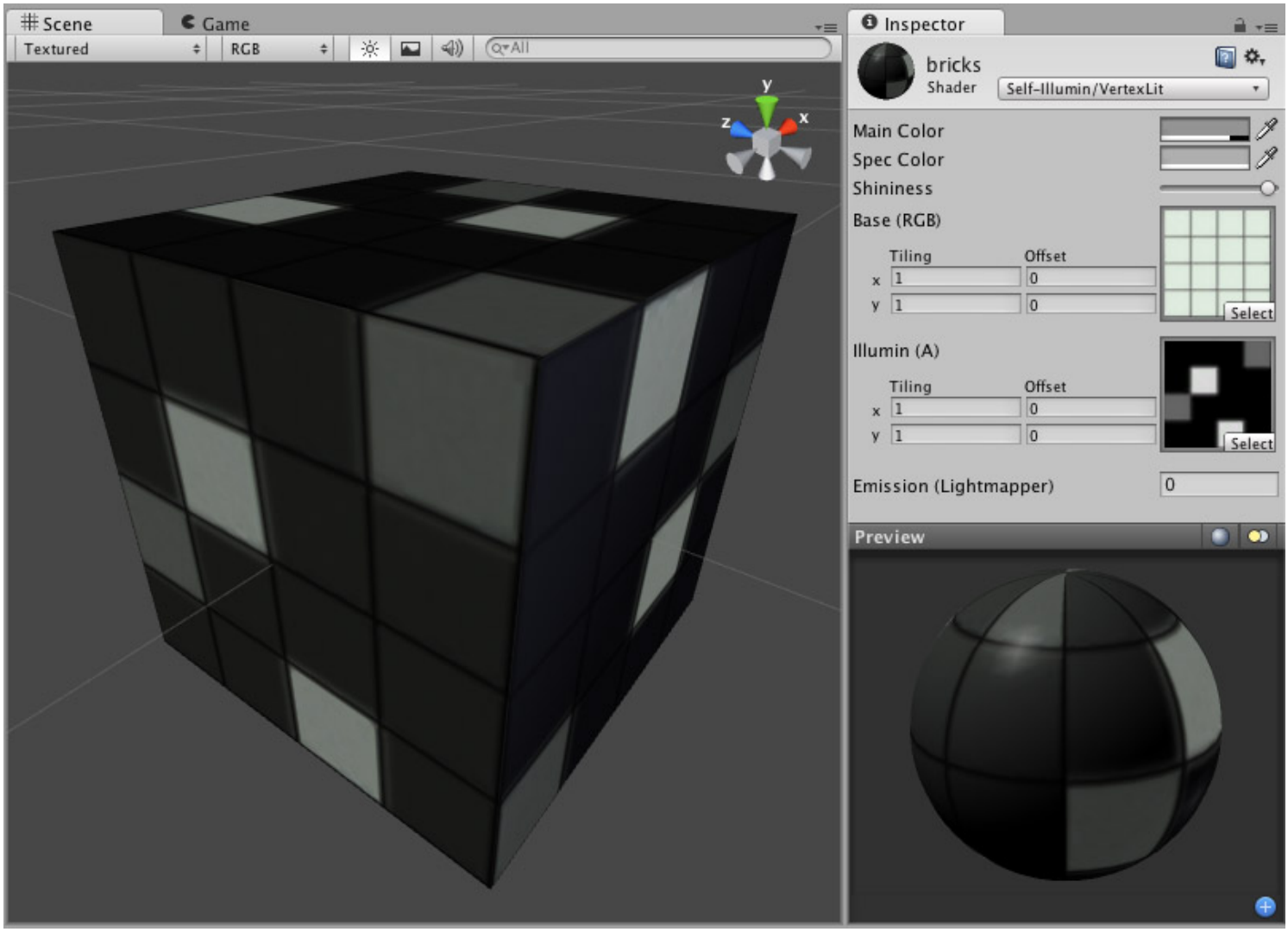
One **Base** texture with alpha channel for Specular Map
One **Normal map** normal map with alpha channel for Illumination Map & Parallax Depth combined
Note: One consideration of this shader is that the **Bumpmap** texture's alpha channel doubles as a Illumination and the Parallax Depth.

» More details

Self-Illuminated Vertex-Lit

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Self-Illuminated Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any **scene** lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your **scene**.

Vertex-Lit Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based **rendering** effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

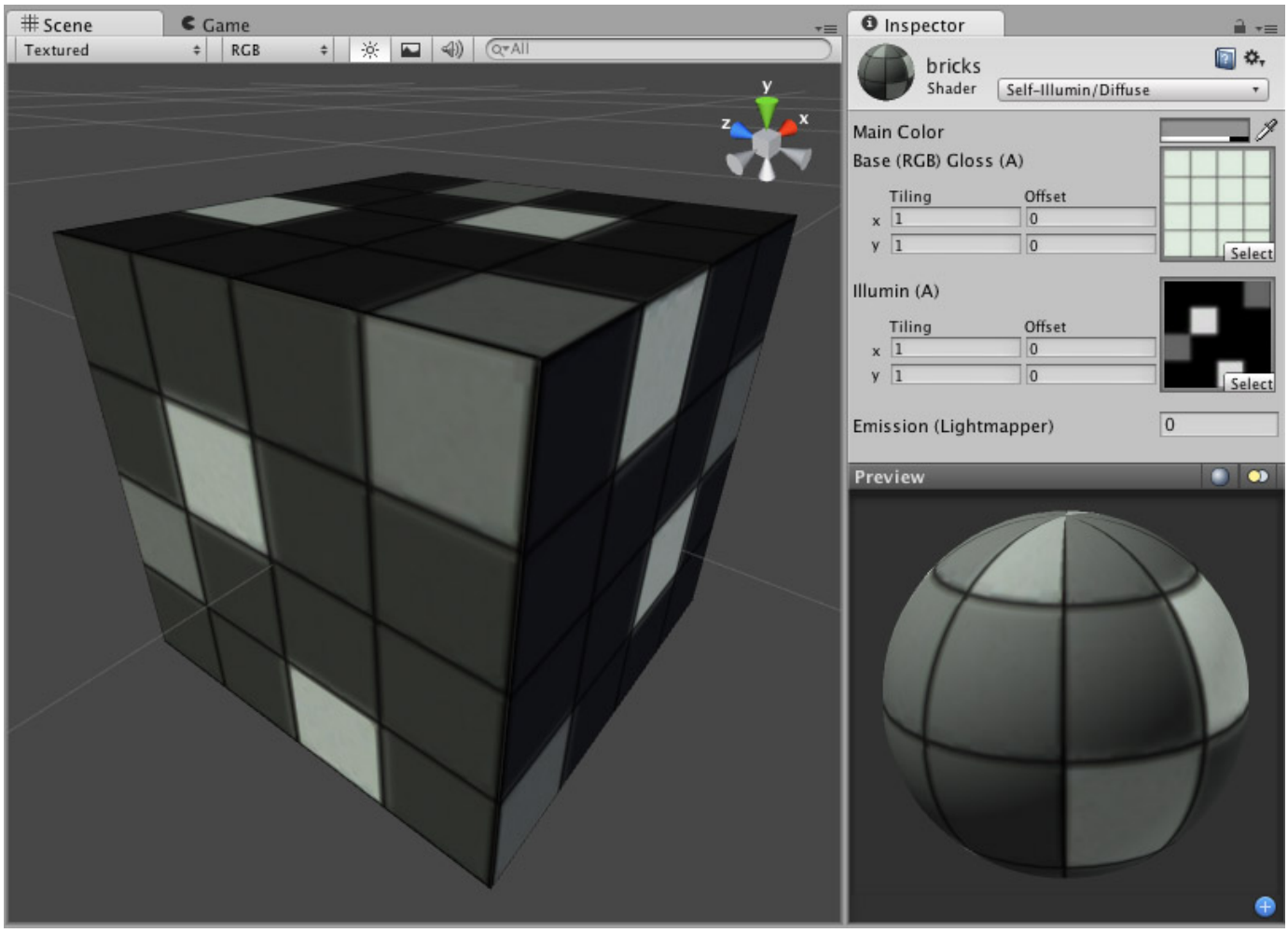
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Self-Illuminated Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Self-Illuminated Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any **scene** lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your **scene**.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the **camera** moves or rotates around.

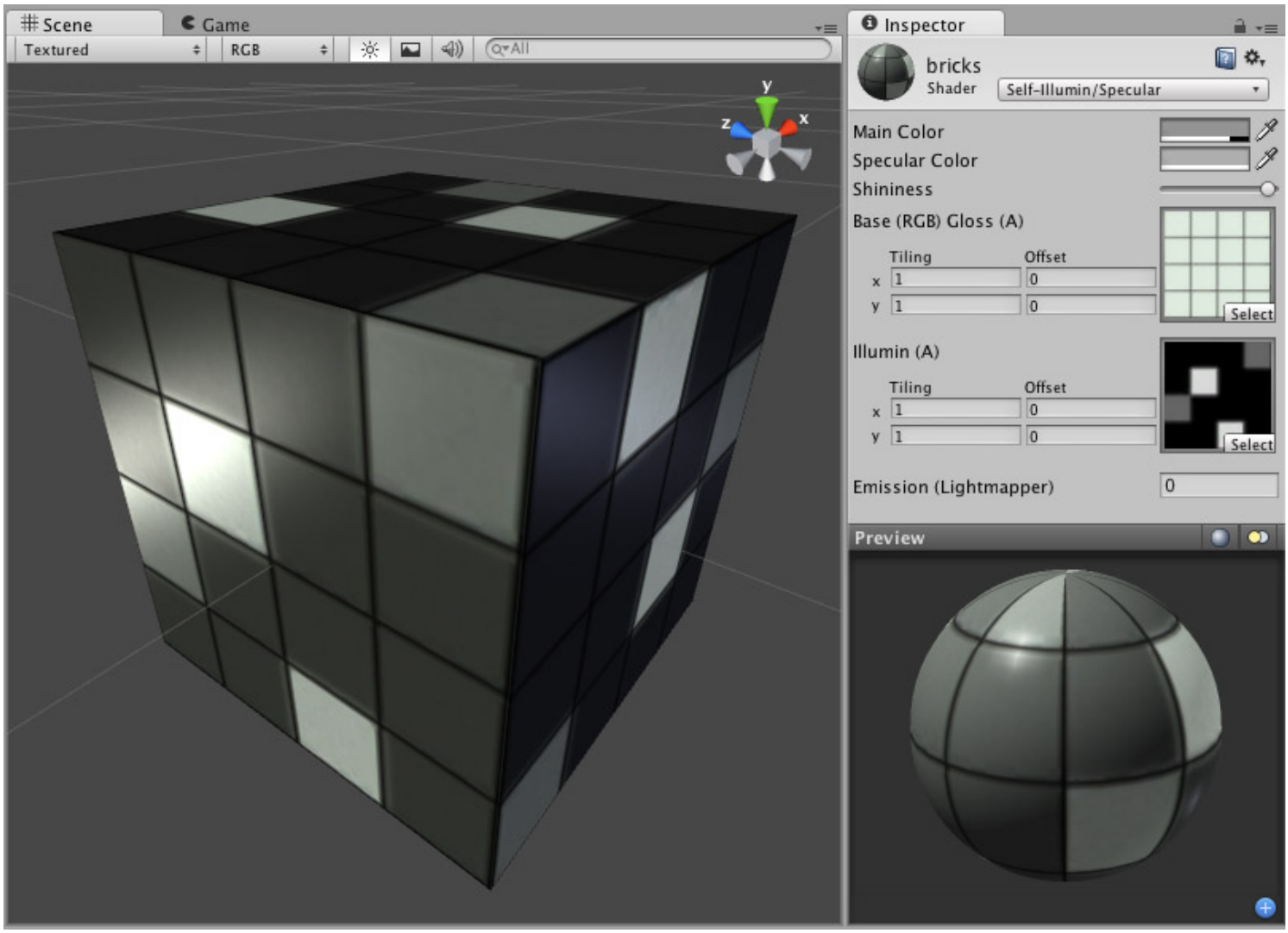
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Self-Illuminated Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Self-Illuminated Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any **scene** lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your **scene**.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different

levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

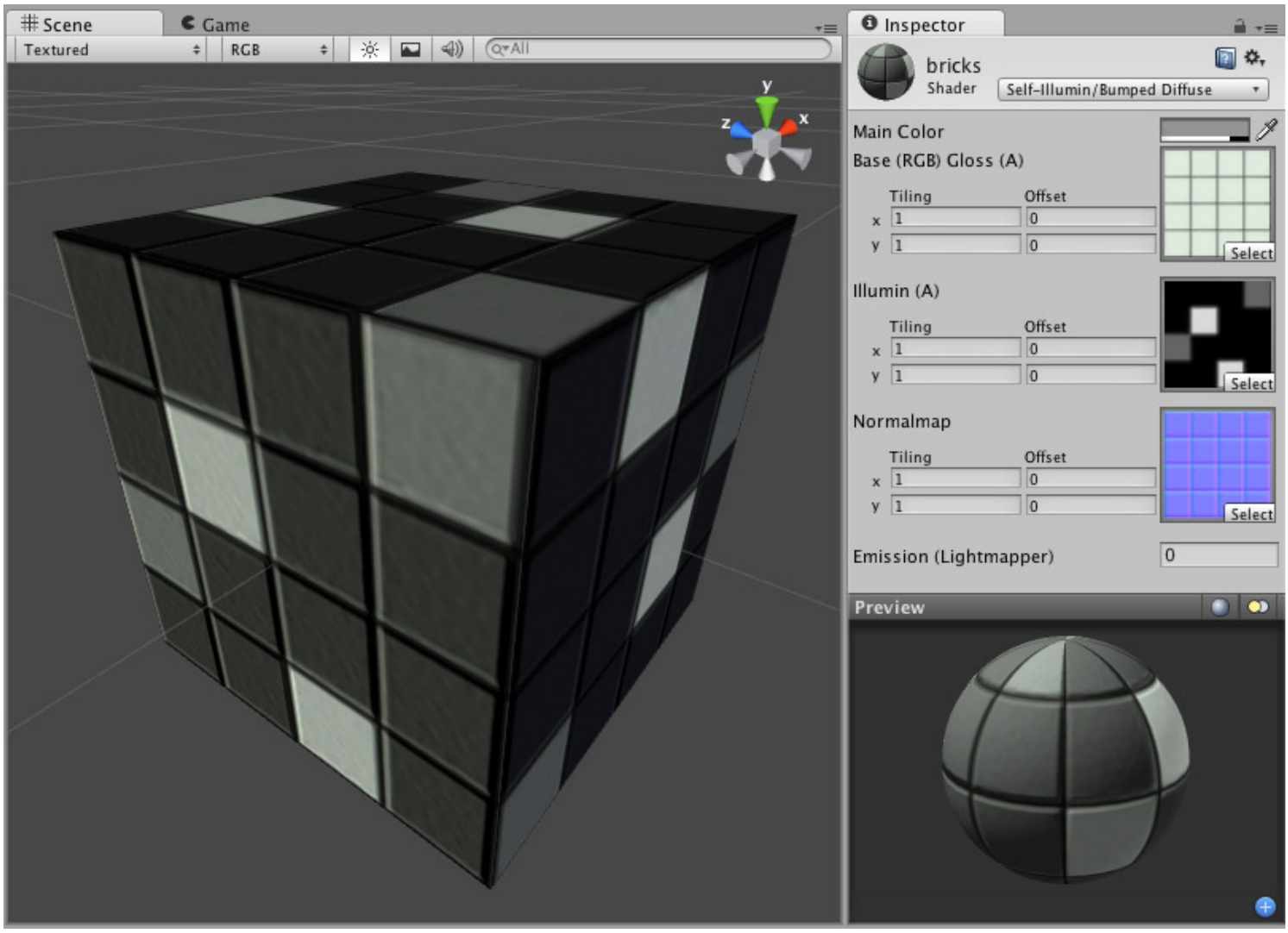
Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Self-Illuminated Normal mapped Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Self-Illuminated Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any **scene** lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your **scene**.

Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the **mesh**'s geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the camera moves or rotates around.

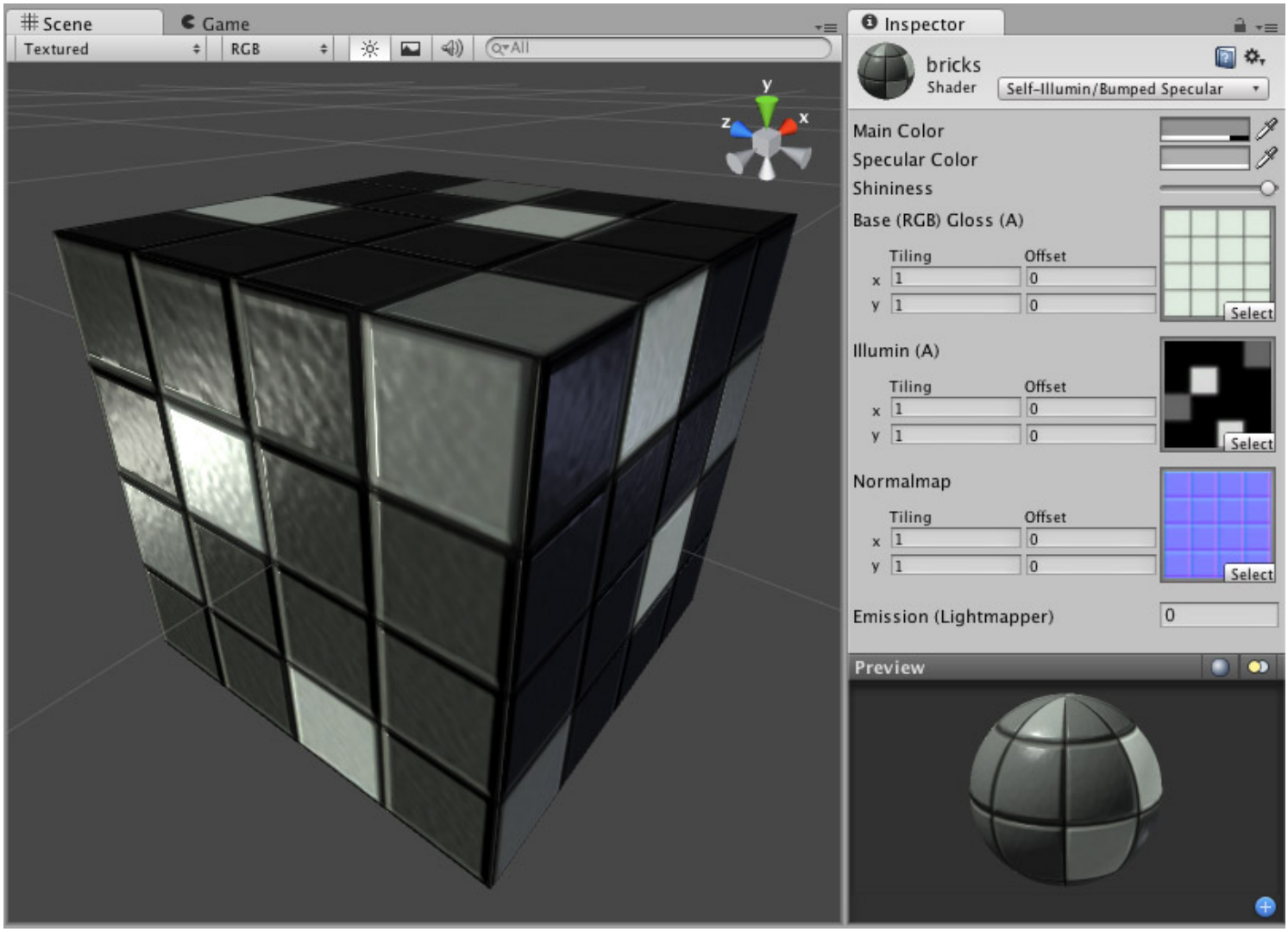
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Self-Illuminated Normal mapped Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Self-Illuminated Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any **scene** lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your **scene**.

Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this

value instead of the one from geometry, lighting is computed. The normal map effectively overrides the **mesh**'s geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

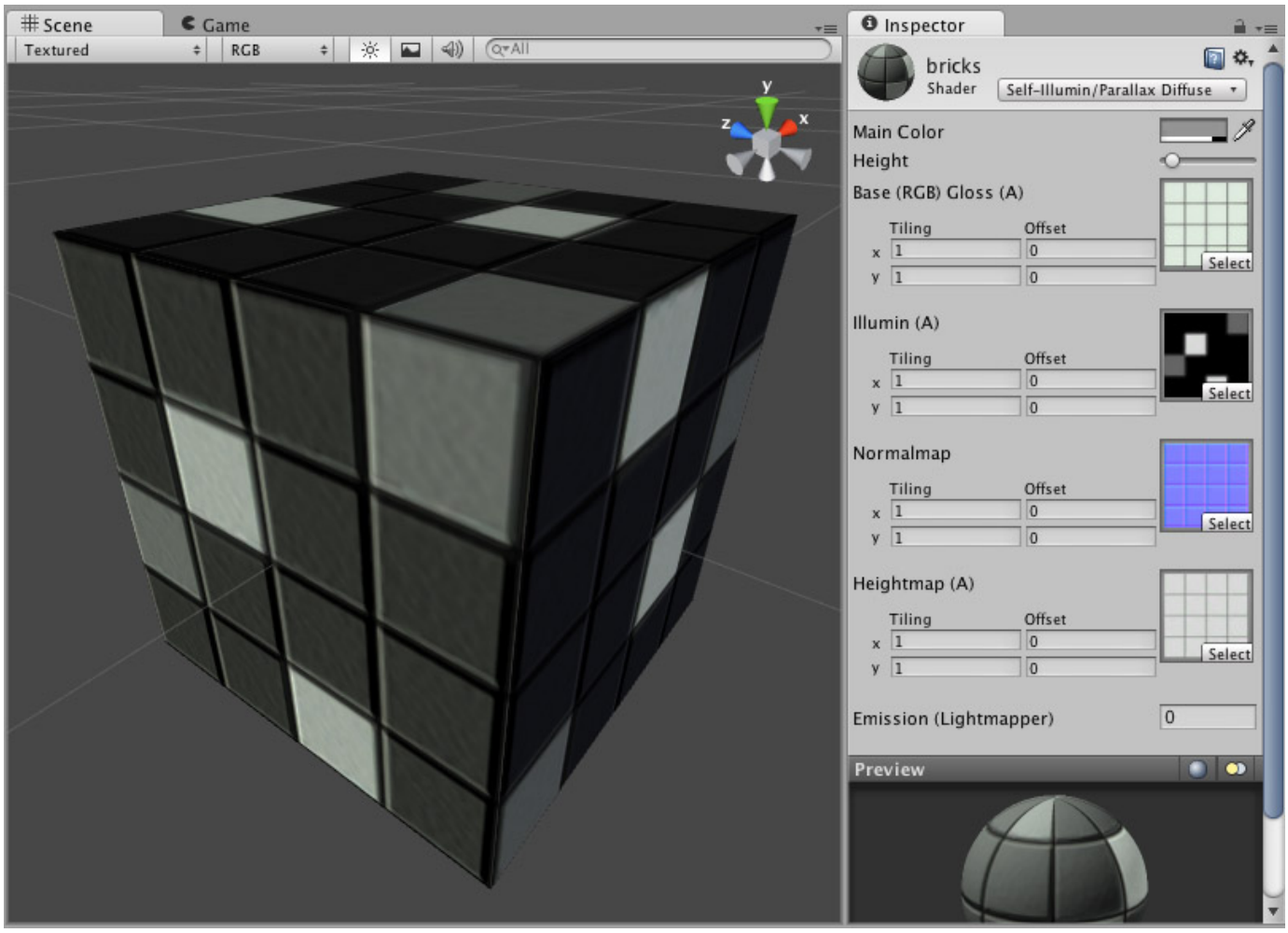
Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Self-Illuminated Parallax Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Self-Illuminated Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any **scene** lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your **scene**.

Parallax Normal mapped Properties

Parallax Normal mapped is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the **camera** moves or rotates around.

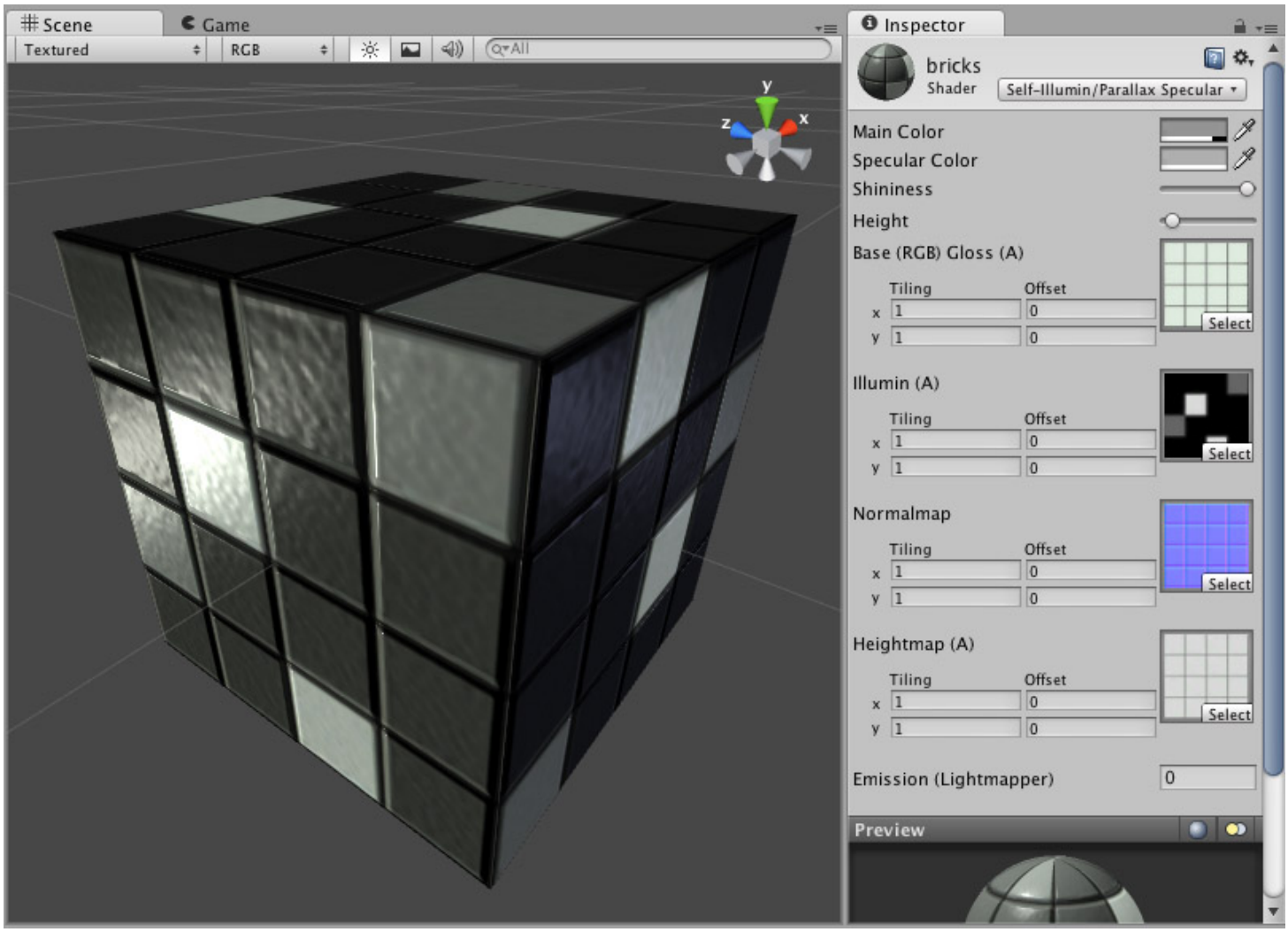
Performance

Generally, this shader is on the more expensive **rendering** side. For more details, please view the Shader Peformance page.

Self-Illuminated Parallax Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Self-Illuminated Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader allows you to define bright and dark parts of the object. The alpha channel of a secondary texture will define areas of the object that "emit" light by themselves, even when no light is shining on it. In the alpha channel, black is zero light, and white is full light emitted by the object. Any **scene** lights will add illumination on top of the shader's illumination. So even if your object does not emit any light by itself, it will still be lit by lights in your **scene**.

Parallax Normal mapped Properties

Parallax Normal mapped is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called “gloss map”), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

Performance

Generally, this shader is on the more expensive **rendering** side. For more details, please view the Shader Peformance page.

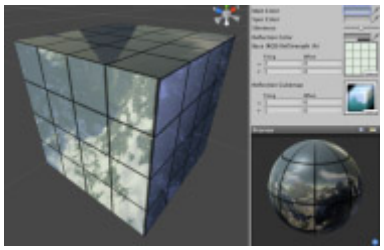
Reflective Shader Family

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces these **shaders**.

Reflective shaders will allow you to use a **Cubemap** which will be reflected on your **mesh**. You can also define areas of more or less reflectivity on your object through the alpha channel of the **Base** texture. High reflectivity is a great effect for glosses, oils, chrome, **etc**. Low reflectivity can add effect for metals, liquid surfaces, or video monitors.

Reflective Vertex-Lit



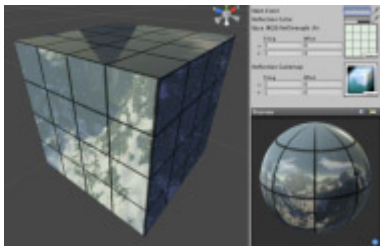
shader-ReflectiveVertexLit

Assets needed:

One **Base** texture with alpha channel for defining reflective areas
One **Reflection** Cubemap for Reflection Map

» More details

Reflective Diffuse



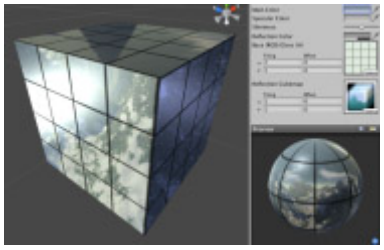
shader-ReflectiveDiffuse

Assets needed:

One **Base** texture with alpha channel for defining reflective areas
One **Reflection** Cubemap for Reflection Map

» More details

Reflective Specular



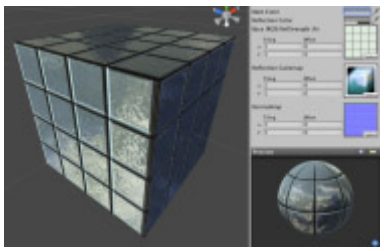
shader-ReflectiveSpecular

Assets needed:

One **Base** texture with alpha channel for defining reflective areas & Specular Map combined
One **Reflection** Cubemap for Reflection Map
Note: One consideration for this shader is that the **Base** texture's alpha channel will double as both the reflective areas and the Specular Map.

» More details

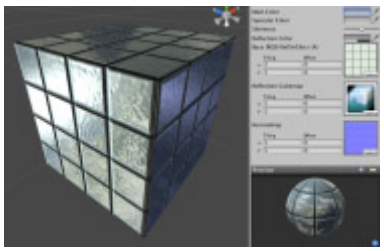
Reflective Normal mapped



shader-ReflectiveBumpedDiffuse
Assets needed:

One **Base** texture with alpha channel for defining reflective areas
One **Reflection** Cubemap for Reflection Map
One **Normal map**, no alpha channel required
» More details

Reflective Normal Mapped Specular

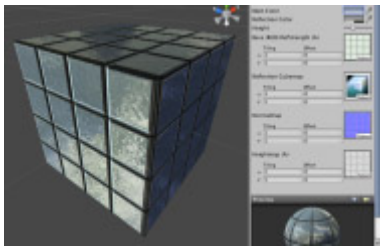


shader-ReflectiveBumpedSpecular
Assets needed:

One **Base** texture with alpha channel for defining reflective areas & Specular Map combined
One **Reflection** Cubemap for Reflection Map
One **Normal map**, no alpha channel required
Note: One consideration for this shader is that the **Base** texture's alpha channel will double as both the reflective areas and the Specular Map.

» More details

Reflective Parallax

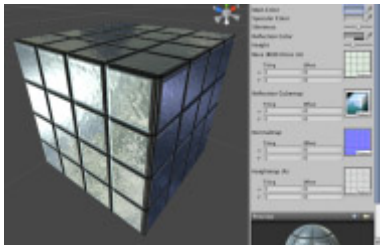


shader-ReflectiveParallaxDiffuse

Assets needed:

One **Base** texture with alpha channel for defining reflective areas
One **Reflection** Cubemap for Reflection Map
One **Normal map**, with alpha channel for Parallax Depth
» More details

Reflective Parallax Specular



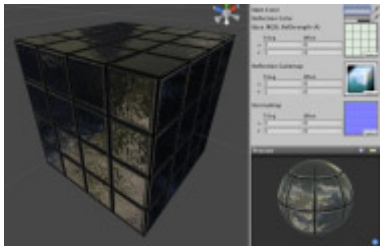
shader-ReflectiveParallaxSpecular

Assets needed:

One **Base** texture with alpha channel for defining reflective areas & Specular Map
One **Reflection** Cubemap for Reflection Map
One **Normal map**, with alpha channel for Parallax Depth
Note: One consideration for this shader is that the **Base** texture's alpha channel will double as both the reflective areas and the Specular Map.

» More details

Reflective Normal mapped Unlit

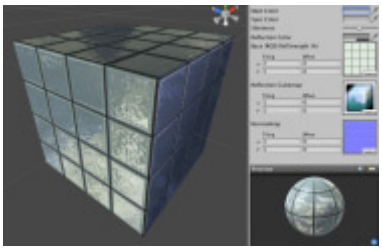


shader-ReflectiveBumpedUnlit

Assets needed:

One **Base** texture with alpha channel for defining reflective areas
One **Reflection** Cubemap for Reflection Map
One **Normal map**, no alpha channel required
» More details

Reflective Normal mapped Vertex-Lit



shader-ReflectiveBumpedVertexLit

Assets needed:

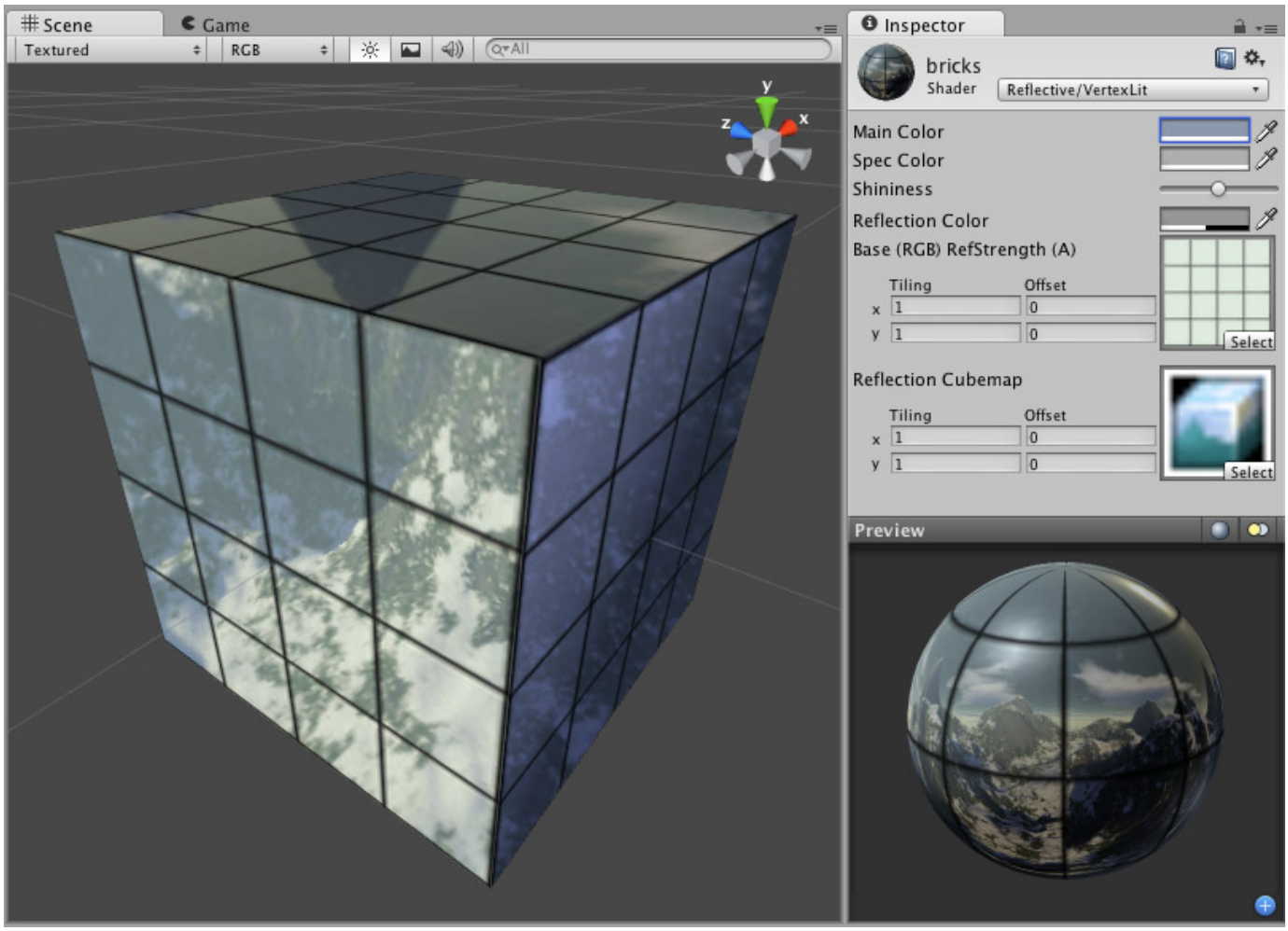
One **Base** texture with alpha channel for defining reflective areas
One **Reflection** Cubemap for Reflection Map
One **Normal map**, no alpha channel required

» More details

Reflective Vertex-Lit

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Reflective Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader will simulate reflective surfaces such as cars, metal objects **etc**. It requires an environment **Cubemap** which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any **scene** lights will add illumination on top of what is reflected.

Vertex-Lit Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based **rendering** effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

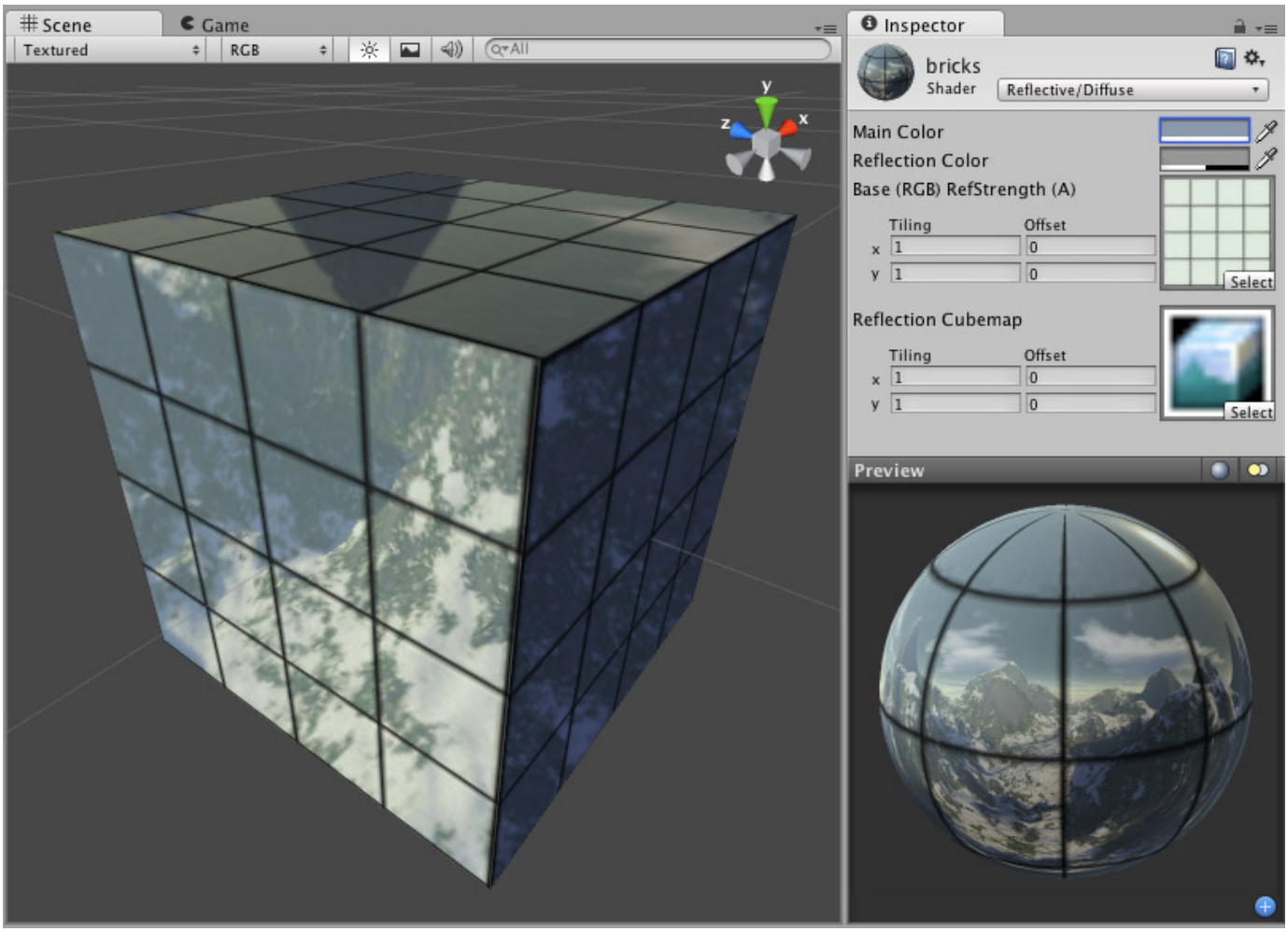
Performance

Generally, this shader is not too expensive to render. For more details, please view the Shader Peformance page.

Reflective Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Reflective Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader will simulate reflective surfaces such as cars, metal objects **etc**. It requires an environment **Cubemap** which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any **scene** lights will add illumination on top of what is reflected.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the **camera** moves or rotates around.

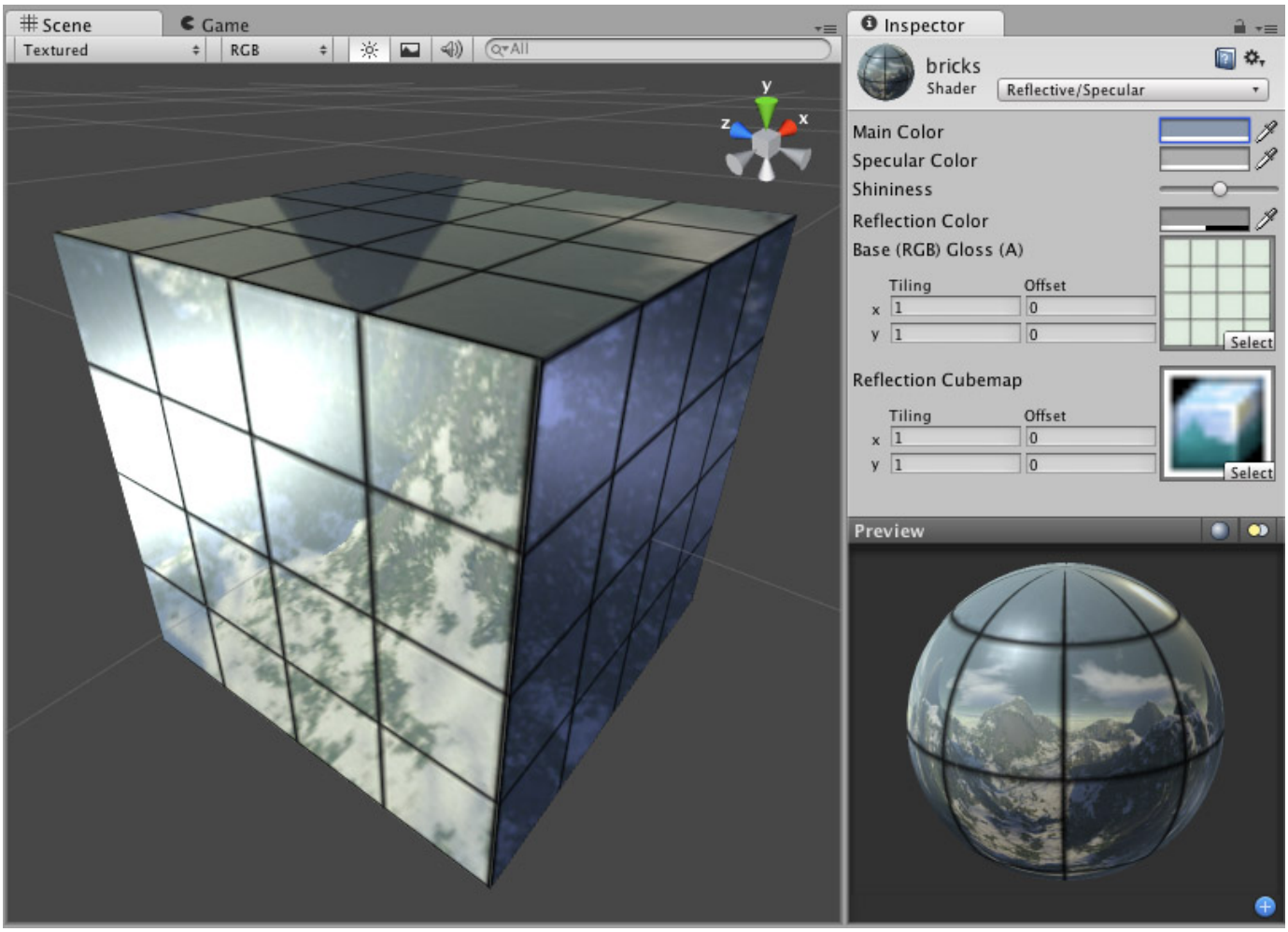
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Reflective Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



One consideration for this shader is that the Base texture's alpha channel will double as both the Reflection Map and the Specular Map.

Reflective Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader will simulate reflective surfaces such as cars, metal objects **etc**. It requires an environment **Cubemap** which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any **scene** lights will add illumination on top of what is reflected.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

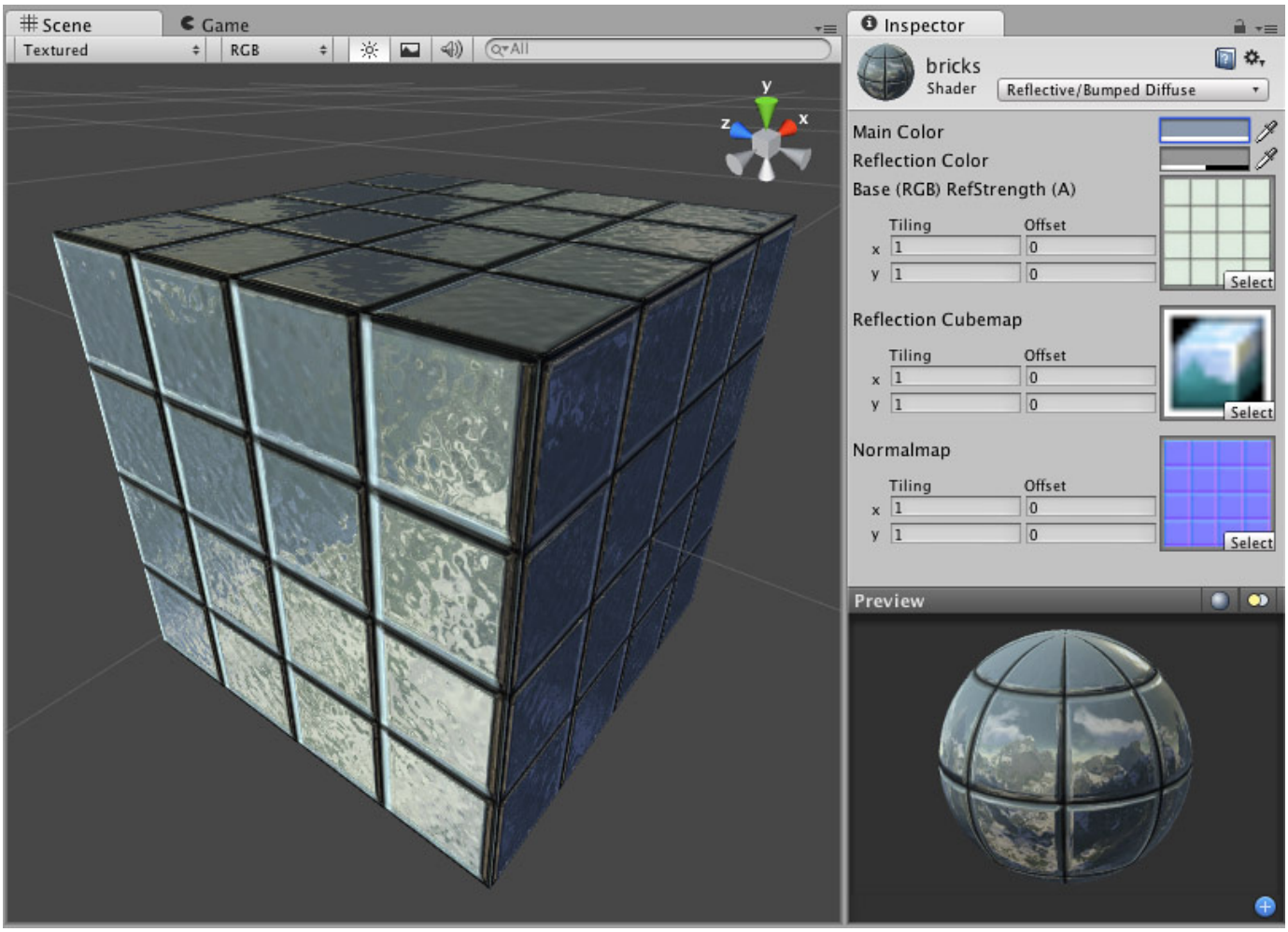
Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Reflective Bumped Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Reflective Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader will simulate reflective surfaces such as cars, metal objects **etc**. It requires an environment **Cubemap** which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any **scene** lights will add illumination on top of what is reflected.

Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the **mesh**'s geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that “follows the surface” of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other “object space” type Normal Maps, but have some advantages:

It’s possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It’s possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the camera moves or rotates around.

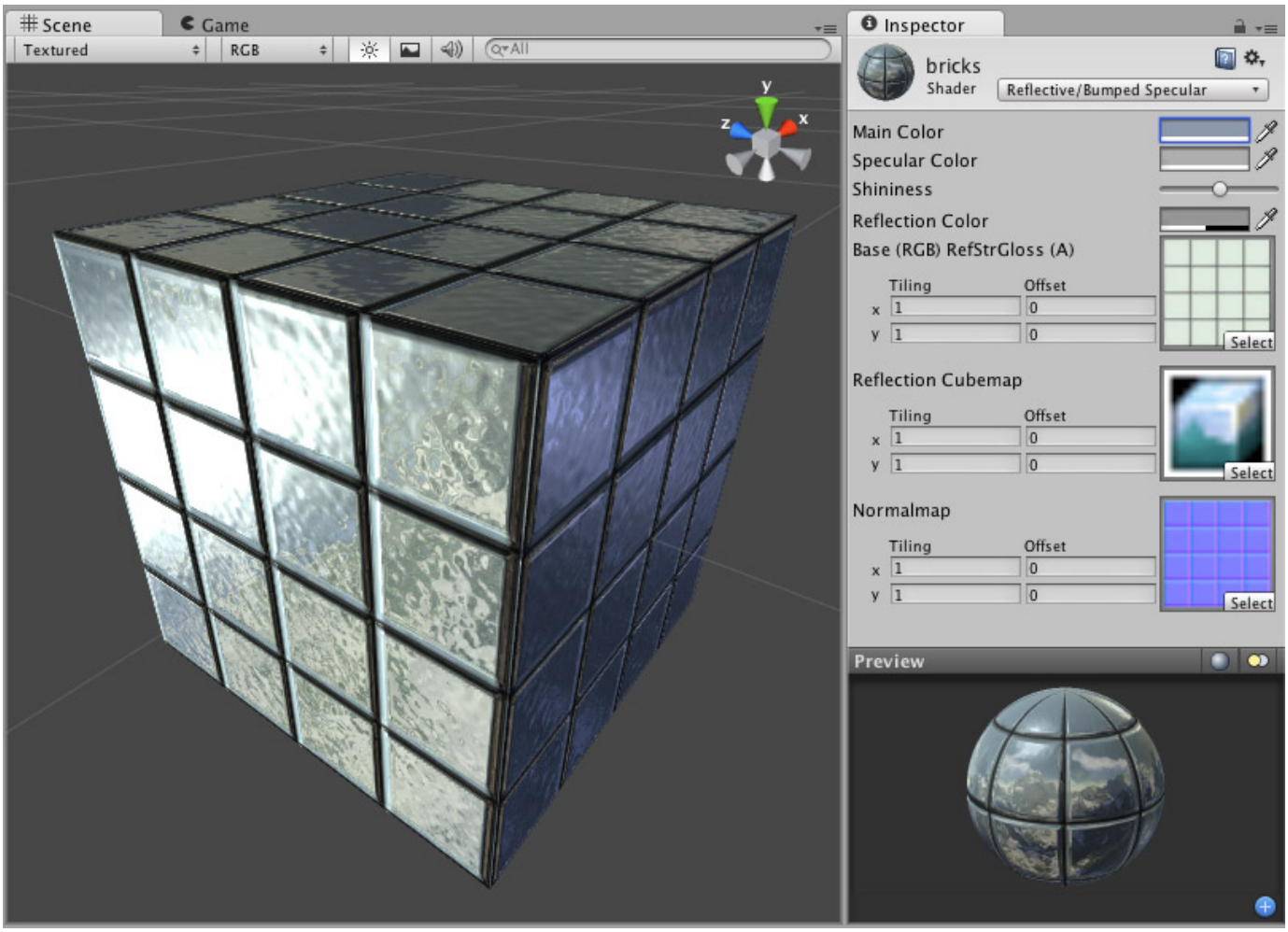
Performance

Generally, this shader is cheap to render. For more details, please view the Shader Peformance page.

Reflective Bumped Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



One consideration for this shader is that the Base texture's alpha channel will double as both the Reflection Map and the Specular Map.

Reflective Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader will simulate reflective surfaces such as cars, metal objects **etc**. It requires an environment **Cubemap** which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any **scene** lights will add illumination on top of what is reflected.

Normal Mapped Properties

Like a **Diffuse** shader, this computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on the angle, and does not change as the **camera** moves or rotates around.

Normal mapping simulates small surface details using a texture, instead of spending more polygons to actually carve out details. It does not actually change the shape of the object, but uses a special texture called a **Normal Map** to achieve this effect. In the normal map, each **pixel**'s color value represents the angle of the surface normal. Then by using this value instead of the one from geometry, lighting is computed. The normal map effectively overrides the **mesh**'s geometry when calculating lighting of the object.

Creating Normal maps

You can import normal maps created outside of Unity, or you can import a regular grayscale image and convert it to a Normal Map from within Unity. (This page refers to a legacy shader which has been superseded by the Standard Shader, but you can learn more about how to use Normal Maps in the Standard Shader)

Technical Details

The Normal Map is a tangent space type of normal map. Tangent space is the space that "follows the surface" of the model geometry. In this space, Z always points away from the surface. Tangent space Normal Maps are a bit more expensive than the other "object space" type Normal Maps, but have some advantages:

It's possible to use them on deforming models - the bumps will remain on the deforming surface and will just work.
It's possible to reuse parts of the normal map on different areas of a model; or use them on different models.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called "gloss map"), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

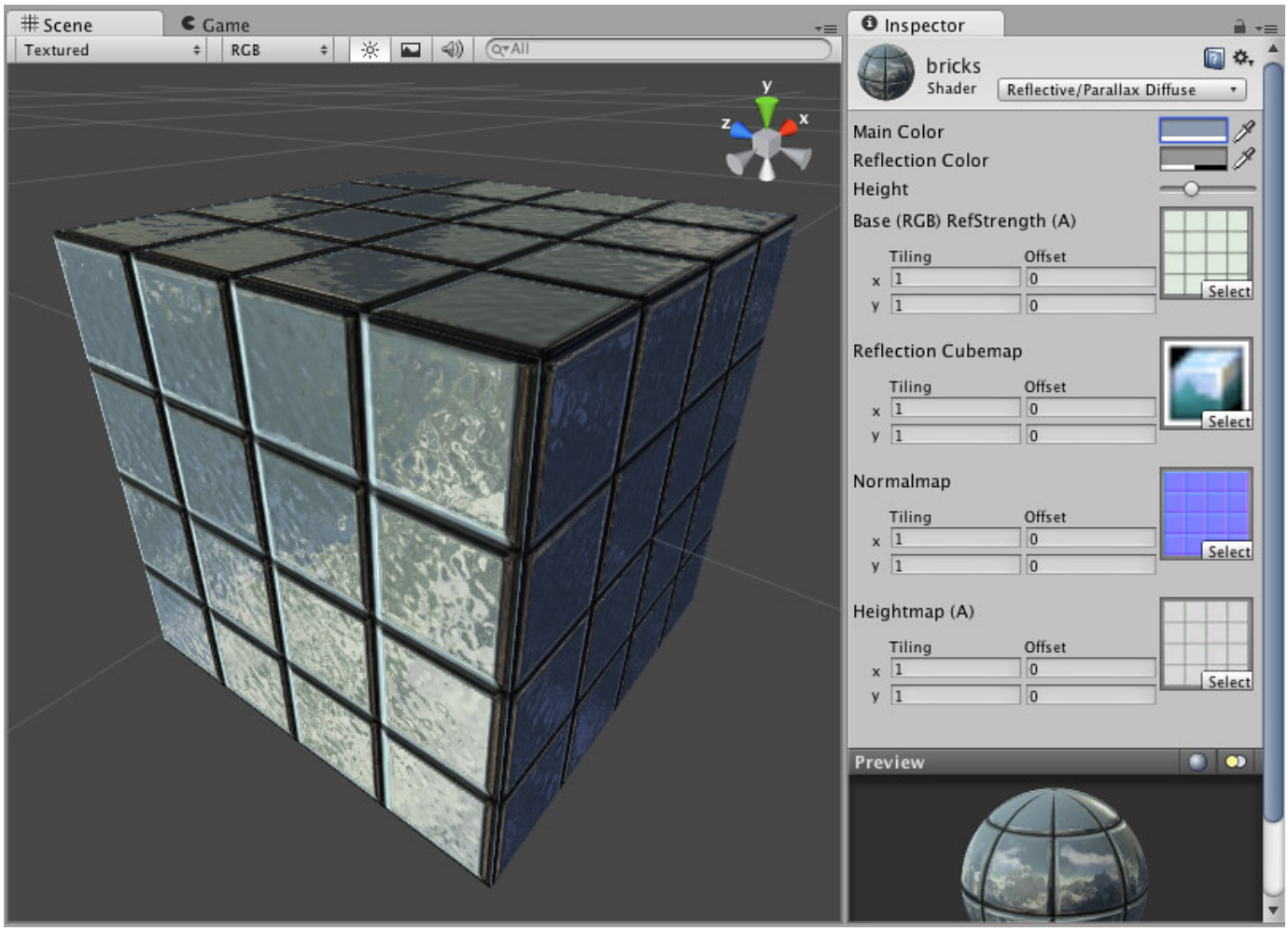
Performance

Generally, this shader is moderately expensive to render. For more details, please view the Shader Peformance page.

Reflective Parallax Diffuse

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Reflective Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader will simulate reflective surfaces such as cars, metal objects **etc**. It requires an environment **Cubemap** which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any **scene** lights will add illumination on top of what is reflected.

Parallax Normal mapped Properties

Parallax Normal mapped is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

Diffuse Properties

Diffuse computes a simple (Lambertian) lighting model. The lighting on the surface decreases as the angle between it and the light decreases. The lighting depends only on this angle, and does not change as the **camera** moves or rotates around.

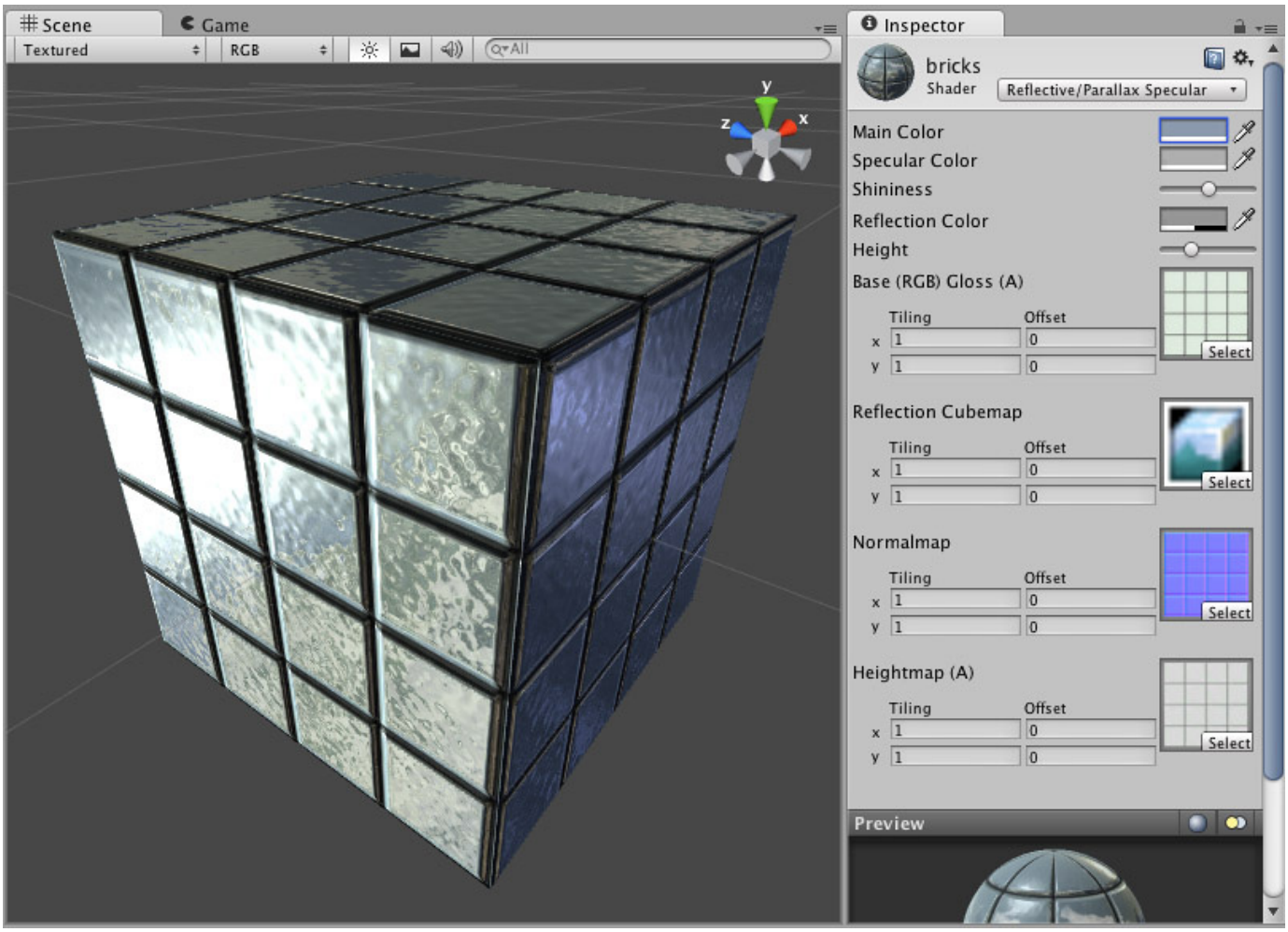
Performance

Generally, this shader is on the more expensive **rendering** side. For more details, please view the Shader Peformance page.

Reflective Parallax Specular

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



One consideration for this shader is that the Base texture's alpha channel will double as both the Reflection Map and the Specular Map.

Reflective Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader will simulate reflective surfaces such as cars, metal objects **etc**. It requires an environment **Cubemap** which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any **scene** lights will add illumination on top of what is reflected.

Parallax Normal mapped Properties

Parallax Normal mapped is the same as regular **Normal mapped**, but with a better simulation of "depth". The extra depth effect is achieved through the use of a **Height Map**. The Height Map is contained in the alpha channel of the Normal map. In the alpha, black is zero depth and white is full depth. This is most often used in bricks/stones to better display the cracks between them.

The Parallax mapping technique is pretty simple, so it can have artifacts and unusual effects. Specifically, very steep height transitions in the Height Map should be avoided. Adjusting the **Height** value in the **Inspector** can also cause the object to become distorted in an odd, unrealistic way. For this reason, it is recommended that you use gradual Height Map transitions or keep the **Height** slider toward the shallow end.

Specular Properties

Specular computes the same simple (Lambertian) lighting as Diffuse, plus a viewer dependent specular highlight. This is called the Blinn-Phong lighting model. It has a specular highlight that is dependent on surface angle, light angle, and viewing angle. The highlight is actually just a realtime-suitable way to simulate blurred reflection of the light source. The level of blur for the highlight is controlled with the **Shininess** slider in the **Inspector**.

Additionally, the alpha channel of the main texture acts as a Specular Map (sometimes called “gloss map”), defining which areas of the object are more reflective than others. Black areas of the alpha will be zero specular reflection, while white areas will be full specular reflection. This is very useful when you want different areas of your object to reflect different levels of specularity. For example, something like rusty metal would use low specularity, while polished metal would use high specularity. Lipstick has higher specularity than skin, and skin has higher specularity than cotton clothes. A well-made Specular Map can make a huge difference in impressing the player.

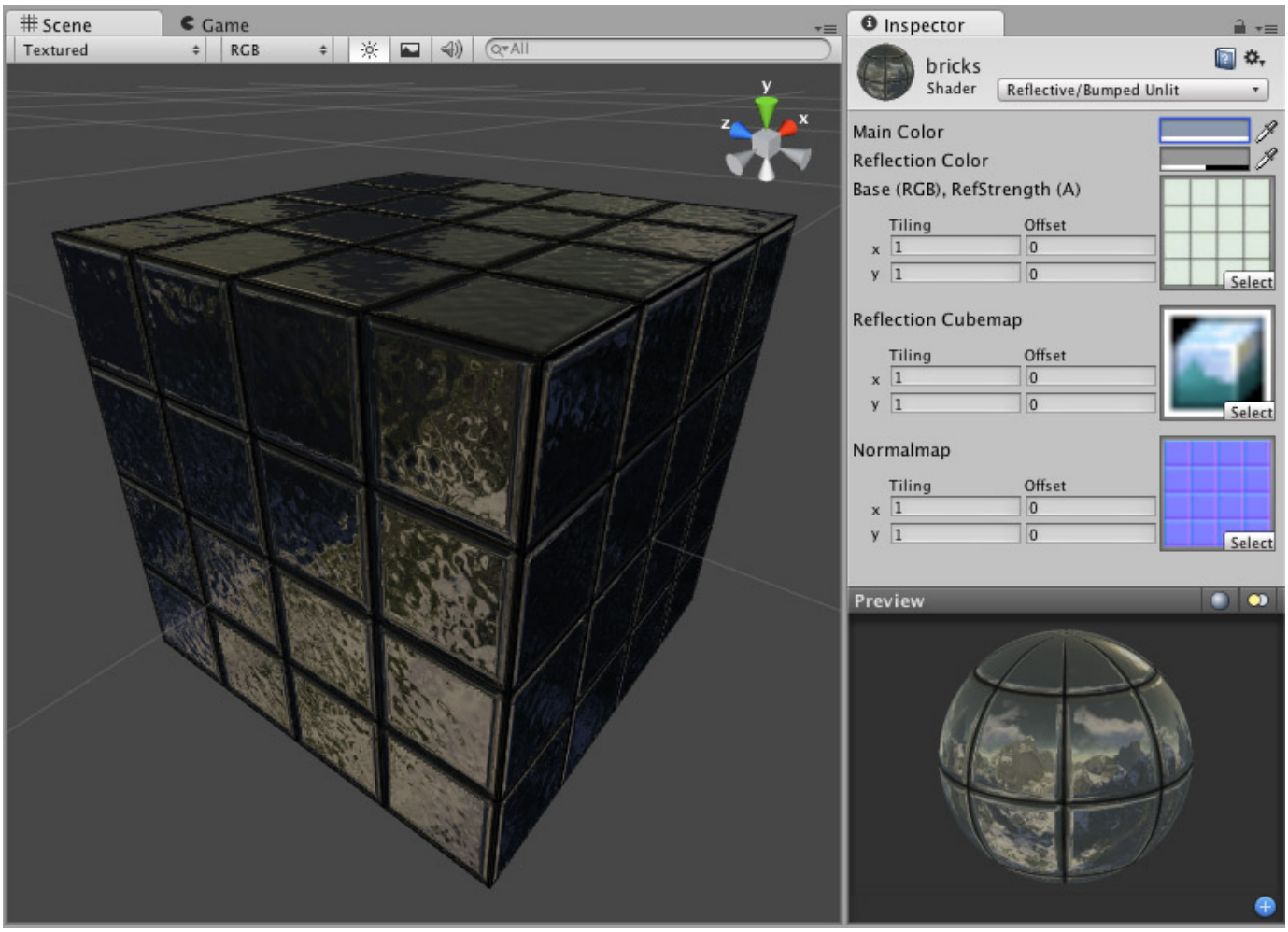
Performance

Generally, this shader is on the more expensive **rendering** side. For more details, please view the Shader Peformance page.

Reflective Normal Mapped Unlit

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Reflective Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader will simulate reflective surfaces such as cars, metal objects **etc**. It requires an environment **Cubemap** which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any **scene** lights will add illumination on top of what is reflected.

Normal mapped Properties

This shader does not use normal-mapping in the traditional way. The normal map does not affect any lights shining on the object, because the shader does not use lights at all. The normal map will only distort the reflection map.

Special Properties

This shader is special because it does not respond to lights at all, so you don't have to worry about performance reduction from use of multiple lights. It simply displays the reflection cube map on the model. The reflection is distorted by the normal map, so you get the benefit of detailed reflection. Because it does not respond to lights, it is quite cheap. It is somewhat of a specialized use case, but in those cases it does exactly what you want as cheaply as possible.

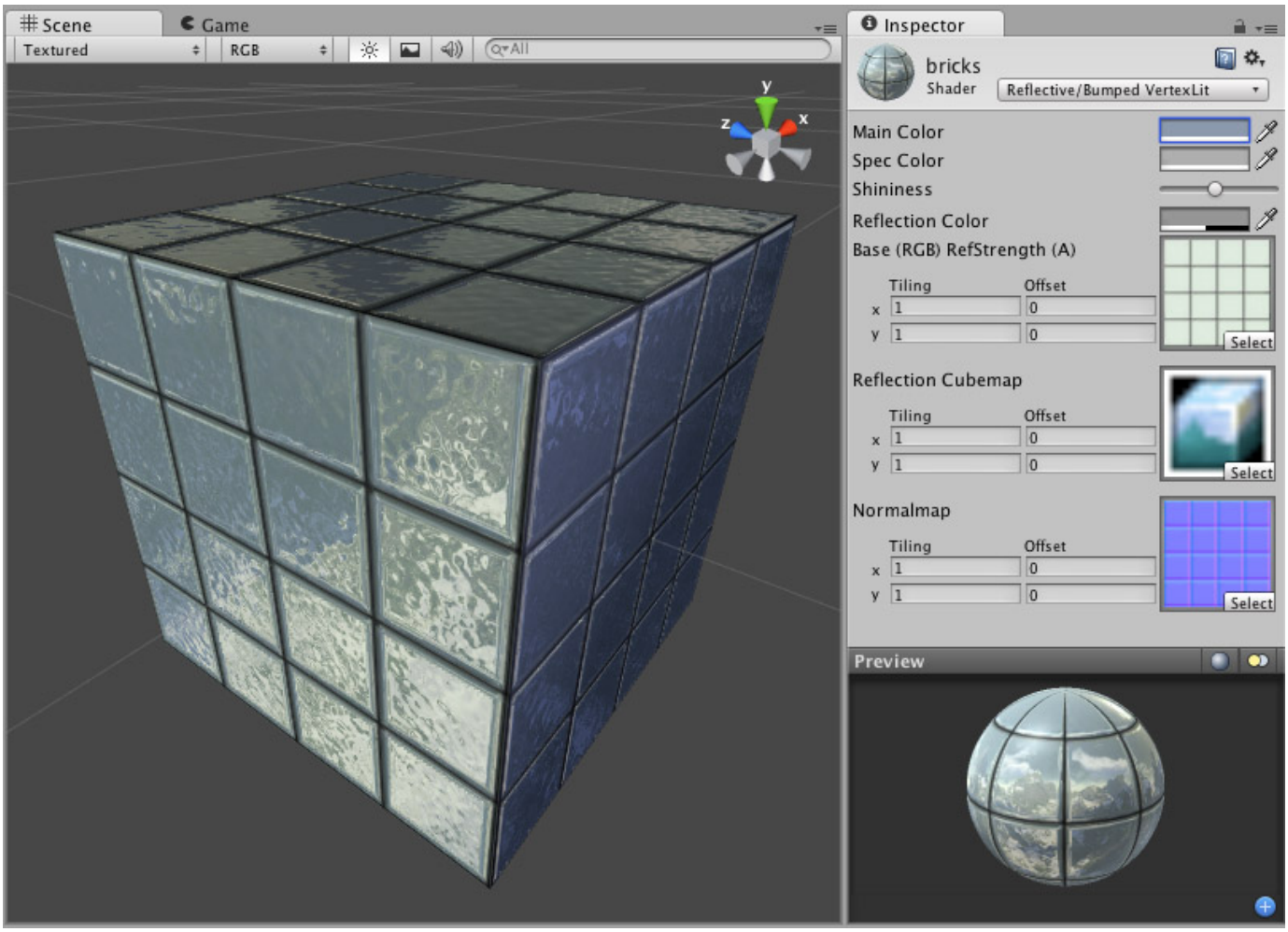
Performance

Generally, this shader is quite cheap to render. For more details, please view the Shader Peformance page.

Reflective Normal mapped Vertex-lit

Leave feedback

Note. Unity 5 introduced the Standard Shader which replaces this **shader**.



Reflective Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader will simulate reflective surfaces such as cars, metal objects **etc**. It requires an environment **Cubemap** which will define what exactly is reflected. The main texture's alpha channel defines the strength of reflection on the object's surface. Any **scene** lights will add illumination on top of what is reflected.

Vertex-Lit Properties

Note. Unity 5 introduced the Standard Shader which replaces this shader.

This shader is **Vertex-Lit**, which is one of the simplest shaders. All lights shining on it are rendered in a single pass and calculated at vertices only.

Because it is vertex-lit, it won't display any pixel-based **rendering** effects, such as light cookies, normal mapping, or shadows. This shader is also much more sensitive to tesselation of the models. If you put a point light very close to a cube using this shader, the light will only be calculated at the corners. Pixel-lit shaders are much more effective at creating a nice round highlight, independent of tesselation. If that's an effect you want, you may consider using a pixel-lit shader or increase tesselation of the objects instead.

Special Properties

This shader is a good alternative to Reflective Normal mapped. If you do not need the object itself to be affected by **pixel** lights, but do want the reflection to be affected by a normal map, this shader is for you. This shader is vertex-lit, so it is rendered more quickly than its Reflective Normal mapped counterpart.

Performance

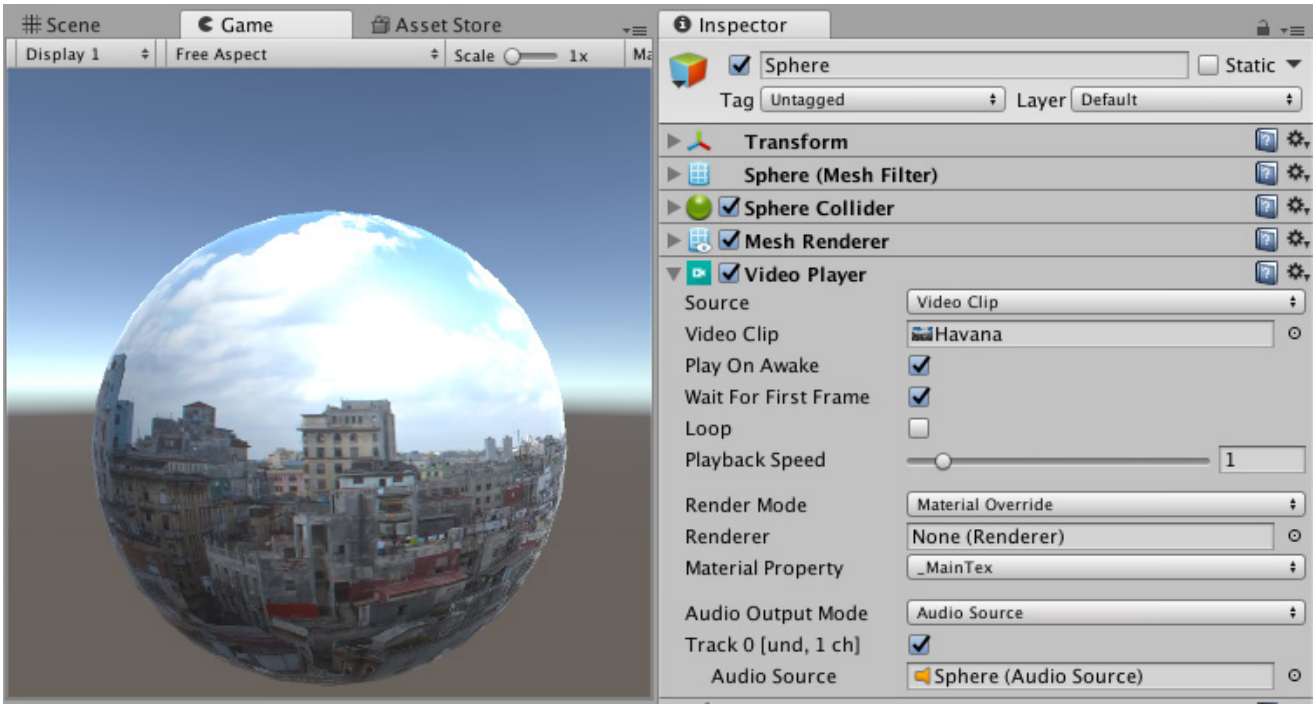
Generally, this shader is not expensive to render. For more details, please view the Shader Peformance page.

Video overview

Leave feedback

Use Unity's video system to integrate video into your game. Video footage can add realism, save on **rendering** complexity, or help you integrate content available externally.

To use video in Unity, import Video Clips and configure them using the Video Player component. The system allows you to feed video footage directly into the **Texture** parameter of any component that has one. Unity then plays the Video on that Texture at run time.



A Video Player component (shown in the Inspector window with a ***Video Clip*** assigned, right) attached to a spherical GameObject (shown in the Game view, left).

Unity's video features include the hardware-accelerated and software decoding of Video files, transparency support, multiple audio tracks, and network streaming.

Note: The Video Player component and Video Clip **asset**, introduced in Unity 5.6, supersede the earlier Movie Textures feature.

2017–06–15 Page published with limited editorial review

Video player support for PS4 added in 2017.1

New feature in Unity 5.6

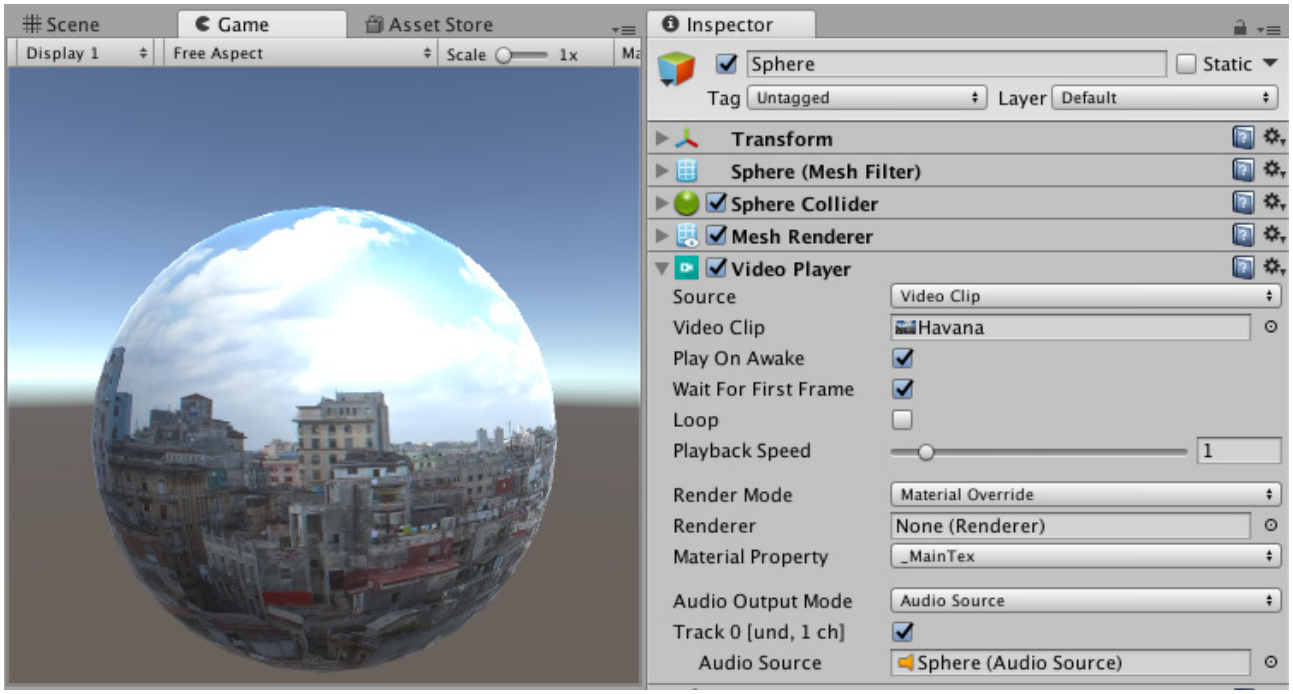
Video Player component

Leave feedback

Use the Video Player component to attach video files to GameObjects, and play them on the GameObject's Texture at run time.

The screenshot below shows a Video Player component attached to a spherical GameObject.

By default, the **Material Property** of a Video Player component is set to *MainTex, which means that when the Video Player component is attached to a GameObject that has a Renderer, it automatically assigns itself to the Texture on that Renderer (because this is the main Texture for the GameObject). Here, the GameObject has a Mesh*_**The main graphics primitive of Unity. Meshes make up a large part of your 3D worlds. Unity supports triangulated or Quadrangulated polygon meshes. Nurbs, Nurms, Subdiv surfaces must be converted to polygons. More info**
See in Glossary Renderer component, so the Video Player automatically assigns it to the Renderer field, which means the Video Clip plays on the Mesh__ Renderer's Texture.



A Video Player component attached to a spherical GameObject, playing the Video Clip on the GameObject's main Texture (in this case, the Texture of the Mesh Renderer)

You can also set a specific target for the video to play on, including:

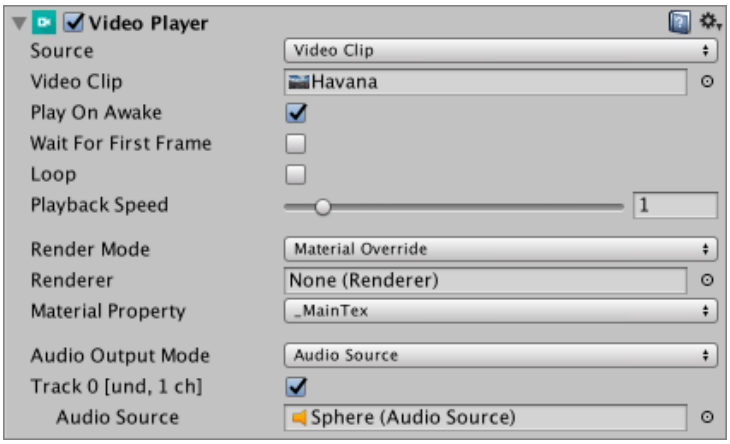
A Camera plane

A Render Texture

A Material Texture parameter

Any Texture field in a component

VideoPlayer component Reference



The Video Player component

Property	*Function*
Source	Choose the type of source for your video.
Video Clip	Assign a Video Clip to the Video Player.
Video Clip	Use this field to define the Video Clip assigned to the Video Player component. Drag-and-drop the video file into this field, or click the circle to the right of the field and choose it from a list of **Assets** if it is in your Project folder.
URL	Assign a video from a URL (for example, http:// or file://). Unity reads the video from this URL at run time.
URL	Enter the URL of the video you want to assign to the Video Player.
Browse...	Click this to quickly navigate your the local file system and open URLs that begin file://.
Play On Awake	Tick the **Play On Awake** checkbox to play the video the moment the **Scene** launches. Untick it if you want to trigger the video playback at another point during run time. Trigger it via scripting with the `Play()` command.
Wait For First Frame	If you tick the **Wait For First Frame** checkbox, Unity waits for the first frame of the source video to be ready for display before the game starts. If you untick it, the first few frames might be discarded to keep the video time in sync with the rest of the game.
Loop	Ticking the **Loop** checkbox to make the Video Player component loop the source video when it reaches its end. If this is unticked, the video stops playing when it reaches the end.
Playback Speed	This slider and numerical field represent a multiplier for the playback speed, as a value between 0 and 10. This is set to 1 (normal speed) by default. If the field is set to 2, the video plays at two times its normal speed.
Render Mode	Use the drop-down to define how the video is rendered.
Camera Far Plane	Render the video on the Camera's far plane.
Camera Near Plane	Render the video on the Camera's near plane.
Camera	Define the Camera receiving the video.
Alpha	The global transparency level added to the source video. This allows elements behind the plane to be visible through it. See documentation on video transparency support for more information about alpha channels.
Render Texture	Render the video into a Render Texture.
Target Texture	Define the Render Texture where the Video Player component renders its images.
Material Override	Render the video into a selected Texture property of a GameObject through its Renderer's Material.
Renderer	The Renderer where the Video Player component renders its images. When set to **None**, the **Renderer** on the same GameObject as the Video Player component is used.

Property	*Function*
Material Property	The name of the Material Texture property that receives the Video Player component images.
API Only	Render the video into the VideoPlayer.texture Scripting API property. You must use scripting to assign the Texture to its intended destination.
Aspect Ratio	The aspect ratio of the images that fill the **Camera Near Plane**, **Camera Far Plane** or **Render Texture** when the corresponding **Render Mode** is used.
No Scaling	No scaling is used. The video is centered on the destination rectangle.
Fit Vertically	Scale the source to fit the destination rectangle vertically, cropping the left and right sides or leaving black areas on each side if necessary. The source aspect ratio is preserved.
Fit Horizontally	Scale the source to fit the destination rectangle horizontally, cropping the top and bottom regions or leaving black areas above and below if needed. The source aspect ratio is preserved.
Fit Inside	Scale the source to fit the destination rectangle without having to crop. Leaves black areas on the left and right or above and below as needed. The source aspect ratio is preserved.
Fit Outside	Scale the source to fit the destination rectangle without leaving black areas on the left and right or above and below, cropping as required. The source aspect ratio is preserved.
Stretch	Scale both horizontally or vertically to fit the destination rectangle. The source aspect ratio is not preserved.
Audio Output Mode	Define how the source's audio tracks are output.
None	Audio is not be played.
Audio Source	Audio samples are sent to selected audio sources, enabling Unity's audio processing to be applied.
Direct	Audio samples are sent directly to the audio output hardware, bypassing Unity's audio processing.
Controlled Tracks	The number of audio tracks in the video. Only shown when **Source** is **URL**. When **Source** is **Video Clip**, the number of tracks is determined by examining the video file.
Track Enabled	When enabled by ticking the relevant checkbox, the associated audio track is used for playback. This must be set prior to playback. The text to the left of the checkbox provides information about the audio track, specifically the track number, language, and number of channels. For example, in the screenshot above, this text is Track 0 [und. 1 ch]. This means it is the first track (Track 0), the language is undefined (und.), and the track has one channel (1 ch), meaning it is a mono track. When the source is a URL, this information is only available during playback. This property only appears if your source is a Video Clip that has an audio track (or tracks), or your source is a URL (allowing you to indicate how many tracks are expected from the URL during playback).
Audio Source	The audio source through which the audio track is played. The targeted audio source can also play Audio Clips. The audio source's playback controls (`Play On Awake` and `Play()` in scripting API) do not apply to the video source's audio track. This property only appears when the **Audio Output Mode** is set to **Audio Source**.

Property	*Function*
Mute	Mute the associated audio track. In **Audio Source** mode, the audio source's control is used. This property only appears when the **Audio Output Mode** is set to **Direct**.
Volume	Volume of the associated audio track. In **Audio Source** mode, the audio source's volume is used. This property only appears when the **Audio Output Mode** is set to **Direct**.

2017–06–15 Page published with limited editorial review

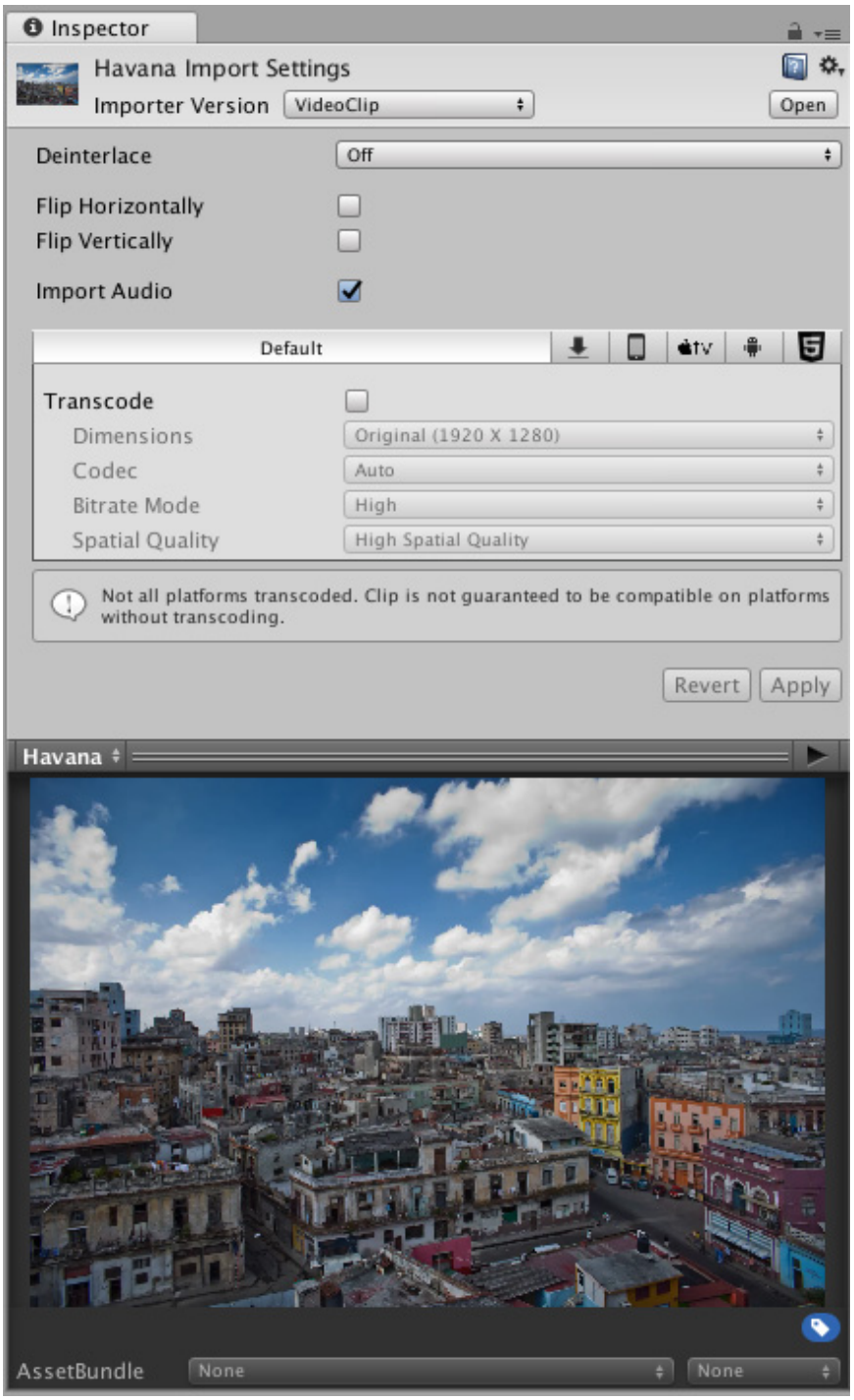
New feature in Unity 5.6

Video Clips

Leave feedback

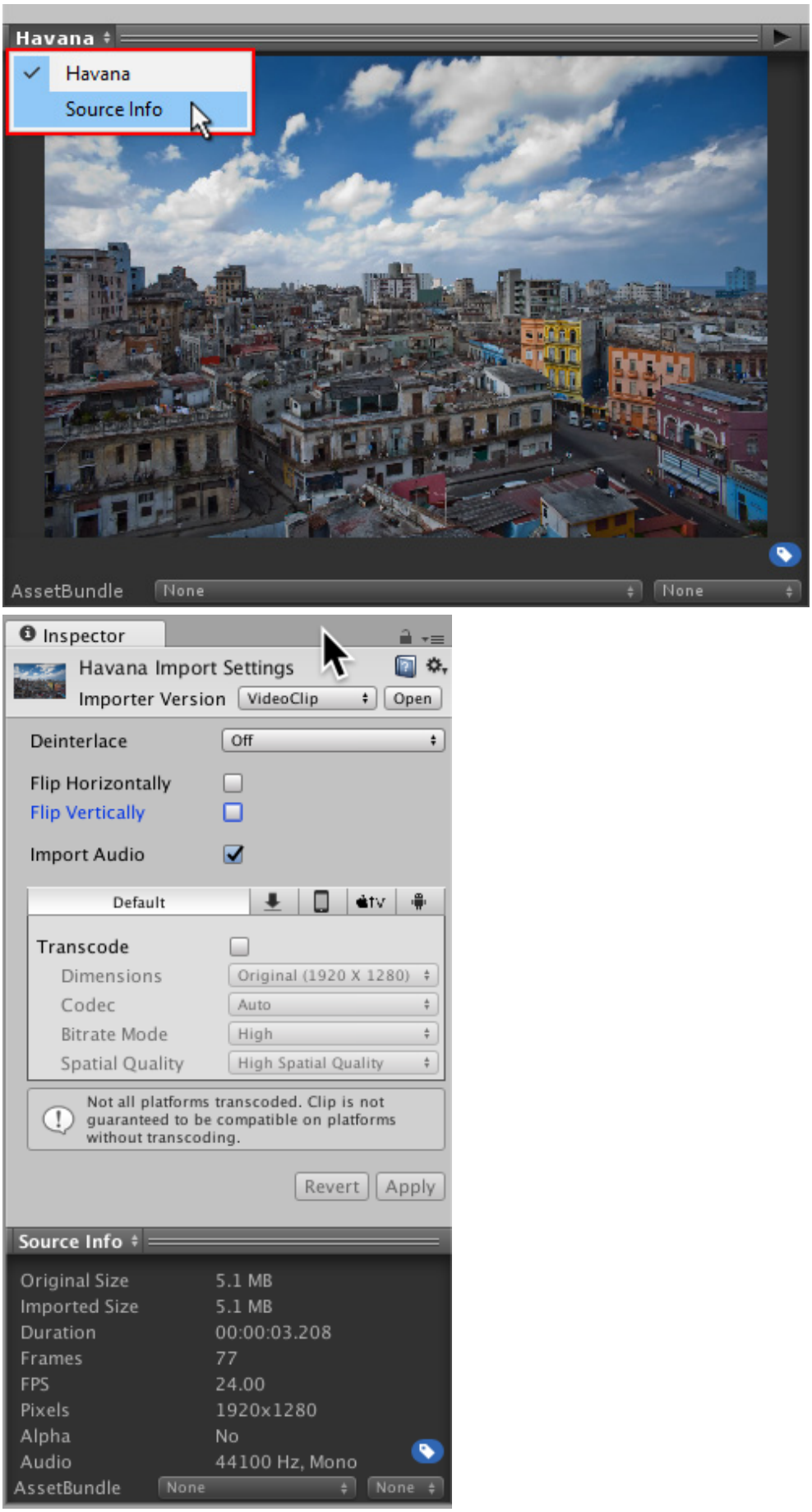
A Video Clip is an imported video file, which the Video Player component uses to play video content (and accompanying audio content, if the video also has sound). Typical file extensions for video files include .mp4, .mov, .webm, and .wmv.

When you select a Video Clip, the **Inspector** shows the Video Clip Importer, including options, preview, and source details. Click the **Play** button at the top-right of the preview to play the Video Clip, along with its first audio track.



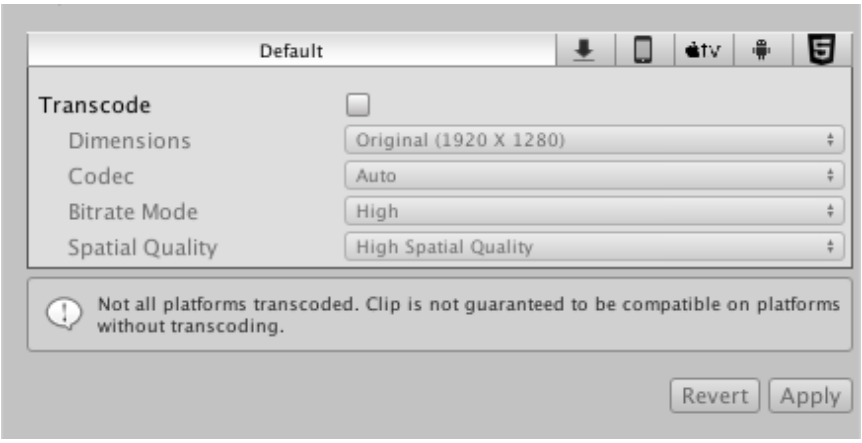
A Video Clip Asset called *Havana* , viewed in the the Inspector window, showing the Video Clip Importer options

To view the source information of a Video Clip, navigate to the preview pane at the bottom of the Inspector window, click the name of the Video Clip in the top-left corner, and select **Source Info**.



The Source Info shows information about the selected Video Clip

Platform-specific options adapt the transcode process for each target platform, allowing the selection of default options for each platform.



The Transcode section of the Video Clip Importer

If transcode is disabled, the video file is used as is, meaning compatibility with the target platform must be verified manually (see documentation on video file compatibility for more information). However, choosing not to transcode can save time as well as avoid transformation-related quality loss.

Video Clip Importer Properties

Property	*Function*
Importer Version	Selects which Importer version to use.
VideoClip	Produces Video Clips, suitable for use with [Video Player components](class-VideoPlayer].
MovieTexture (Legacy)	Produces legacy Movie Textures.
Keep Alpha	Keeps the alpha channel and encodes it during transcode so it can be used even if the target platform does not natively support videos with alpha. This property is only visible for sources that have an alpha channel.
Deinterlace	Controls how interlaced sources are deinterlaced during transcode. For example, you can choose to change interlacing settings due to how a video is encoded or for aesthetic reasons. Interlaced videos have two time samples in each frame: one in odd-numbered lines, and one in even-numbered lines.
Off	The source is not interlaced, and there is no processing to do (this is the default setting).
Even	Takes the even lines of each frame and interpolates them to create missing content. Odd lines are dropped.
Odd	Takes the odd lines of each frame and interpolates them to create missing content. Even lines are dropped.
Flip Horizontally	If this tickbox is checked, the source content is flipped along a horizontal axis during transcode to make it upside-down.
Flip Vertically	If this tickbox is checked, the source content is flipped along a vertical axis during transcode to swap left and right.
Import Audio	If this tickbox is checked, audio tracks are imported during transcode. This property is only visible for sources that have audio tracks.

Property	*Function*
Transcode	When enabled by ticking the checkbox, the source is transcoded into a format that is compatible with the target platform. If disabled, the original content is used, bypassing the potentially lengthy transcoding process. **Note:** Verify that the source format is compatible with each target platform (see documentation on video file compatibility for more information).
Dimensions	Controls how the source content is resized.
Original	Keeps the original size.
Three Quarter Res	Resizes the source to three quarters of its original width and height.
Half Res	Resizes the source to half of its original width and height.
Quarter Res	Resizes the source to a quarter of its original width and height.
Square (1024 X 1024)	Resizes the source to 1024 x 1024 square images. Aspect ratio is controllable.
Square (512 X 512)	Resizes the source to 512 x 512 square images. Aspect ratio is controllable.
Square (256 X 256)	Resizes the source to to 256 x 256 square images. Aspect ratio is controllable.
Custom	Resizes the source to a custom resolution. Aspect ratio is controllable.
Width	Width of the resulting images. This property is only visible when the Dimensions field is set to Custom.
Height	Height of the resulting images. This property is only visible when the Dimensions field is set to Custom.
Aspect Ratio	Aspect ratio control used when resizing images. This property is only visible when the Dimensions field is set to an option other than Original.
No Scaling	Black areas added as needed to preserve the aspect ratio of the original content.
Stretch	Stretches the original content to fill the destination resolution without leaving black areas.
Codec	Codec used for encoding the video track.
Auto	Chooses the most suitable video codec for the target platform.
H264	The MPEG–4 AVC video codec, supported by hardware on most platforms.
VP8	The VP8 video codec, supported by software on most platforms, and by hardware on a few platforms such as Android and **WebGL**.
Bitrate Mode	Low, Medium, or High bitrate, relative to the chosen codec's baseline profile.
Spatial Quality	This setting dictates whether video images are reduced in size during transcoding, which means they take up less storage space. However, resizing images also results in blurriness during playback.

Property	*Function*
Low Spatial Quality	The image is significantly reduced in size during transcode (typically to a quarter of its original dimensions), and then expanded back to its original size upon playback. This is the highest amount of resizing, meaning it saves the most storage space but results in the largest amount of blurriness upon playback.
Medium Spatial Quality	The image is moderately reduced in size during transcode (typically to half of its original dimensions), and then expanded back to its original size upon playback. Although there is some resizing, images will be less blurry than those that use the Low Spatial Quality option, and there is some reduction in required storage space.
High Spatial Quality	No resizing takes place if this option is selected. This means the image is not reduced in size during transcode, and therefore the video's original level of visual clarity is maintained.

2017–06–15 Page published with limited editorial review

New feature in Unity 5.6

Video sources

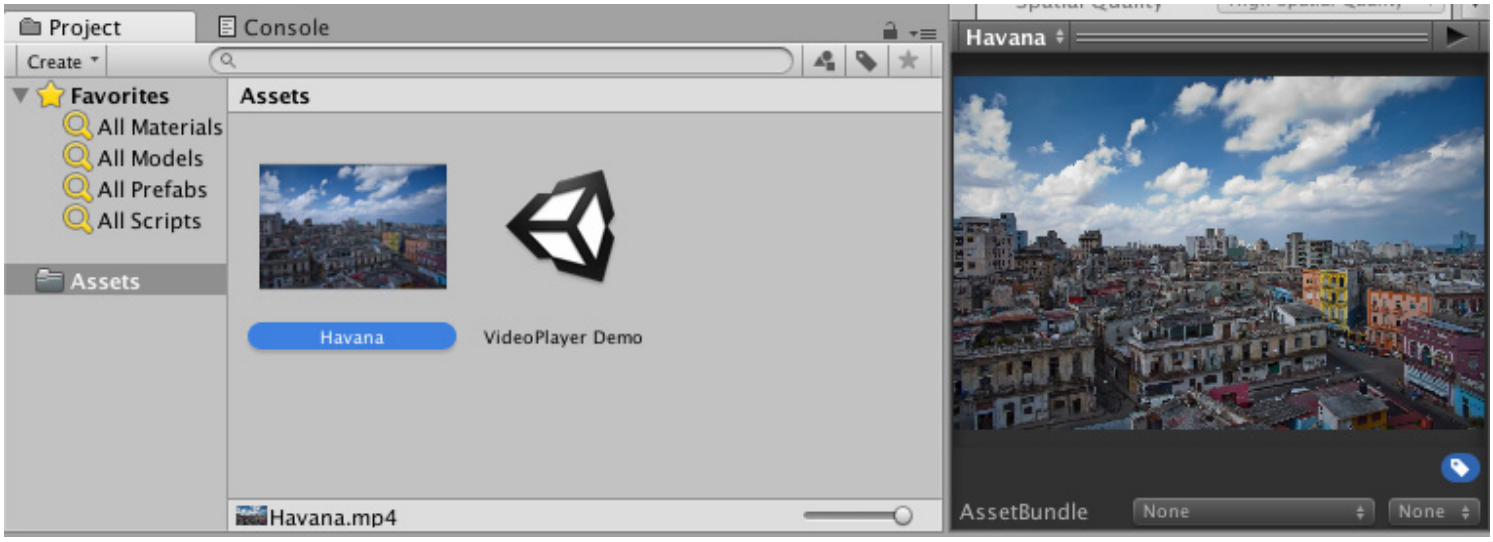
Leave feedback

The Video Player component can play content imported from a variety of sources.

Video Clip

To create and use a Video Clip Asset, you must first import a video file.

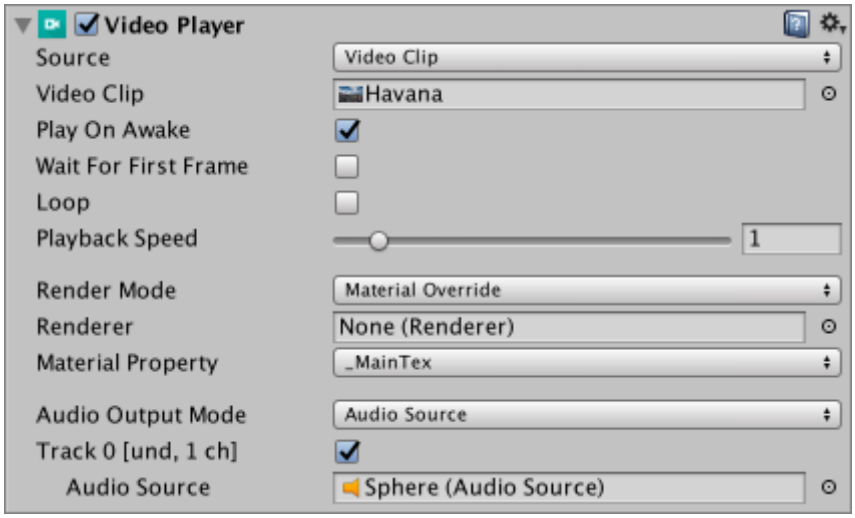
Dragging and dropping a video file into the Project window creates a Video Clip.



A Video Clip created by dragging and dropping a video file into the Project window

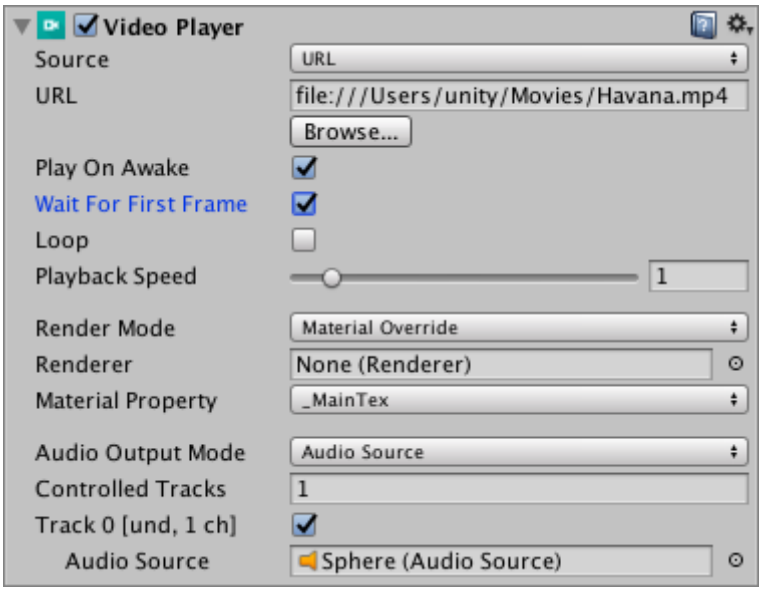
Another way to create a Video Clip is to navigate to **Assets** > **Import New Asset...** to import a video file.

Once imported, the newly created Video Clip can be selected in the Video Player **component** using either the **Select VideoClip** window (accessible by clicking the circle select button to the right of the Video Clip field) or by dragging and dropping the Video Clip Asset into the corresponding Video Player component field.



A **Video Player** component

URLs



The Source field in the Video Player component

Use the drop-down **Source** menu to set the video source to **URL** (this property is set to **Video Clip** by default). Setting **Source** to **URL** makes it possible to directly use files in the filesystem, with or without a *file://* prefix.

The **URL** source option bypasses Asset management, meaning you must manually ensure that the source video can be found by Unity. For example, a web URL needs a web server to host the source video, while a normal file must be located somewhere where Unity can find it, indicated with scripting. However, this can be useful for situations where the content is not under Unity's direct control, or if you want to avoid storing large video files locally.

Setting the Video Player component source to **URL** can also be used to read videos from web sources via *http://* and *https://*. In these cases, Unity performs the necessary pre-buffering and error management.

Asset Bundles

Video Clips can also be read from Asset Bundles.

Once imported, these Video Clips can be used by assigning it to the Video Player component's **Video Clip** field.

Streaming Assets

Files placed in Unity's StreamingAssets folder can be used via the Video Player component's **URL** option (see above), or by using the platform-specific path (*Application.streamingAssetsPath*).

2017–06–15 Page published with limited editorial review

New feature in Unity 5.6

Video file compatibility

Leave feedback

Unity can import video files of many different formats. After import a video file is stored as a VideoClip **asset**.

However, the compatibility of these varies according to which platform you are using - see the table below for a full compatibility list.

Extension	Windows	OSX	Linux
.asf	✓		
.avi	✓		
.dv	✓	✓	
.m4v	✓	✓	
.mov	✓	✓	
.mp4	✓	✓	
.mpg	✓	✓	
.mpeg	✓	✓	
.ogv	✓	✓	✓
.vp8	✓	✓	✓
.webm	✓	✓	✓
.wmv	✓		

Each of these formats can contain tracks with many different [codecs]. Each version of each platform also supports a different list of codecs, so make sure to consult the official documentation for the platform you are working on. For example, Windows and OSX both provide official documentation on their respective codec compatibility - see official Windows and OSX documentation for further compatibility information about these platforms.

If the Editor is unable to read the content of a track within a file, it produces an error message. If this happens, you must convert or re-encode the track from the source so it is usable by your Editor platform's native libraries.

H.264 (typically in a .mp4, .m4v, or .mov format) is the optimal supported video codec because it offers the best compatibility across platforms.

Video Clips can also be used without transcoding by unchecking the **Transcode** checkbox in the Video Clip importer (see documentation on [video sources] for more information). This allows you to use video files without any additional conversion, which is faster and prevents quality loss due to re-encoding.

Note: For best results, make sure to check that your video files are supported on each target platform.

2017–06–15 Page published with limited editorial review

New feature in Unity 5.6

Understanding video files

Leave feedback

Video files are more accurately described as “containers”. This is because they can contain not only the video itself but also additional tracks including audio, subtitles, and further video footage. There can also be more than one of each type of track in a container, for example:

Multiple points of view

Stereo or 5.1 versions of the audio mix

Subtitles in different languages

Dialogue in different languages

To save on bandwidth and storage, each track’s content is encoded using a “codec”, which compresses and decompresses data as required.

A common video codec format is H.264, and a common audio codec format is AAC.

File extensions such as .mp4, .mov, .webm or .avi indicate that the data in the video file is arranged using a certain container format.

Hardware and software decoding

Most modern devices have hardware dedicated to decoding videos. This hardware typically requires less power to do this task than, for example, the CPU, and means that the resources can be used for tasks other than decoding videos.

This hardware acceleration is made possible by native custom APIs, which vary from platform to platform. Unity’s video architecture hides these differences by providing a common **UI** and Scripting API in order to access these capabilities.

Unity is also capable of software-based video decoding. This uses the VP8 video codec and Vorbis audio codec, and is useful for situations where a platform’s hardware decoding results in unwanted restrictions in terms of resolution, the presence of multiple audio tracks, or support of alpha channel (see documentation on Transparency for more information).

2017–06–15 Page published with limited editorial review

New feature in Unity 5.6

Video transparency support

Leave feedback

Unity's Video Clips and Video Player component support alpha, which is the standard term used to refer to transparency.

In graphics terminology, "alpha" is another way of saying "transparency". Alpha is a continuous value, not something that can be switched on or off.

The lowest alpha value means an image is fully transparent (not visible at all), while the highest alpha value means it is fully opaque (the image is solid and cannot be seen through). Intermediate values make the image partially transparent, allowing you to see both the image and the background behind it simultaneously.

The Video Player **component** supports a global alpha value when playing its content in a **Camera**'s near or far planes. However, videos can have per-pixel alpha values, meaning that transparency can vary across the video image. This per-pixel transparency control is done in applications that produce images and videos (such as NUKE or After Effects), and not in the Unity Editor.

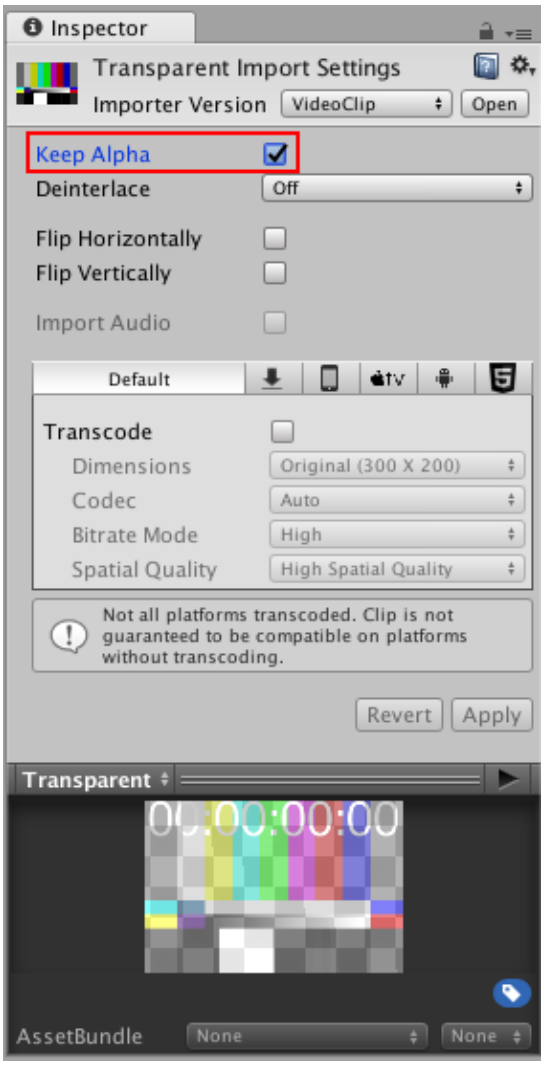
Unity supports two types of sources that have per-pixel alpha:

Apple ProRes 4444

The Apple ProRes 4444 codec is an extremely high-quality version of Apple ProRes for 4:4:4:4 image sources, including alpha channels. It offers the same level of visual fidelity as the source video.

Apple ProRes 4444 is only supported on OSX because this is the only platform where it is available natively. It normally appears in .mov files.

When importing a video that uses this codec, enable both the **Transcode** and **Keep Alpha** options by ticking the relevant checkboxes in the Video Clip Importer. Your operating system's video playback software may have the functionality to identify which codecs your video uses.



A Video Clip Asset viewed in the Inspector, showing the ***Keep Alpha*** option - highlighted in red - enabled

During transcoding, Unity inserts the alpha into the color stream so it can be used both with H.264 or VP8.

Omitting the transcode operation leaves the ProRes representation in the **Asset**, meaning the target platform has to support this codec (see documentation on video file compatibility for more information).

This codec also usually produces large files, which increases storage and bandwidth requirements.

Webm with VP8

The .webm file format has a specification refinement that allows it to carry alpha information natively when combined with the VP8 video codec. This means any Editor platform can read videos with transparency with this format.

Because most of Unity's supported platforms use a software implementation for decoding these files, they don't need to be transcoded for these platforms.

One notable exception is Android. This platform's native VP8 support does not include transparency support, which means transcoding must be enabled so Unity uses its internal alpha representation.

2017–06–15 Page published with limited editorial review

New feature in Unity 5.6

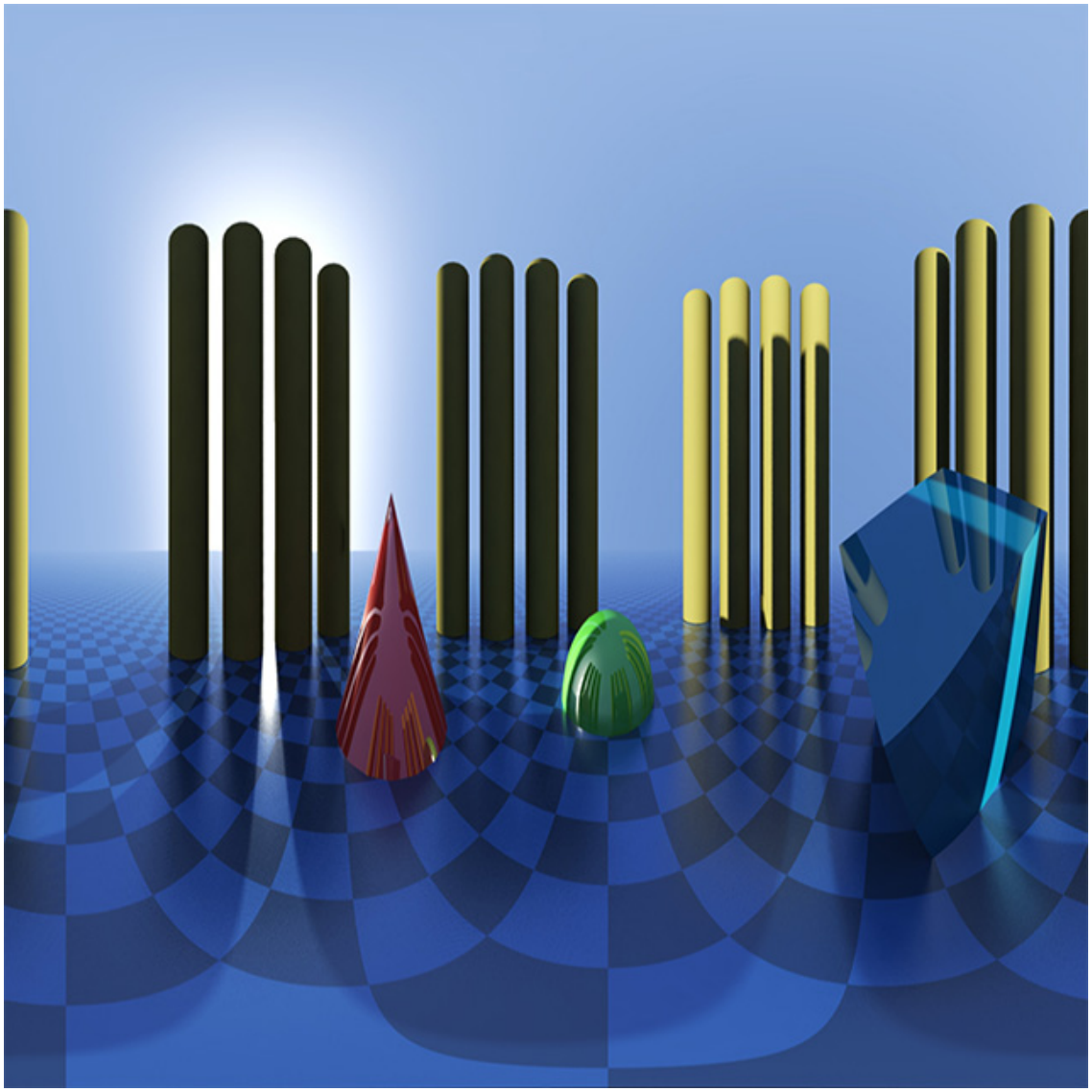
Panoramic video

Leave feedback

Unity's panoramic video feature enables you to:

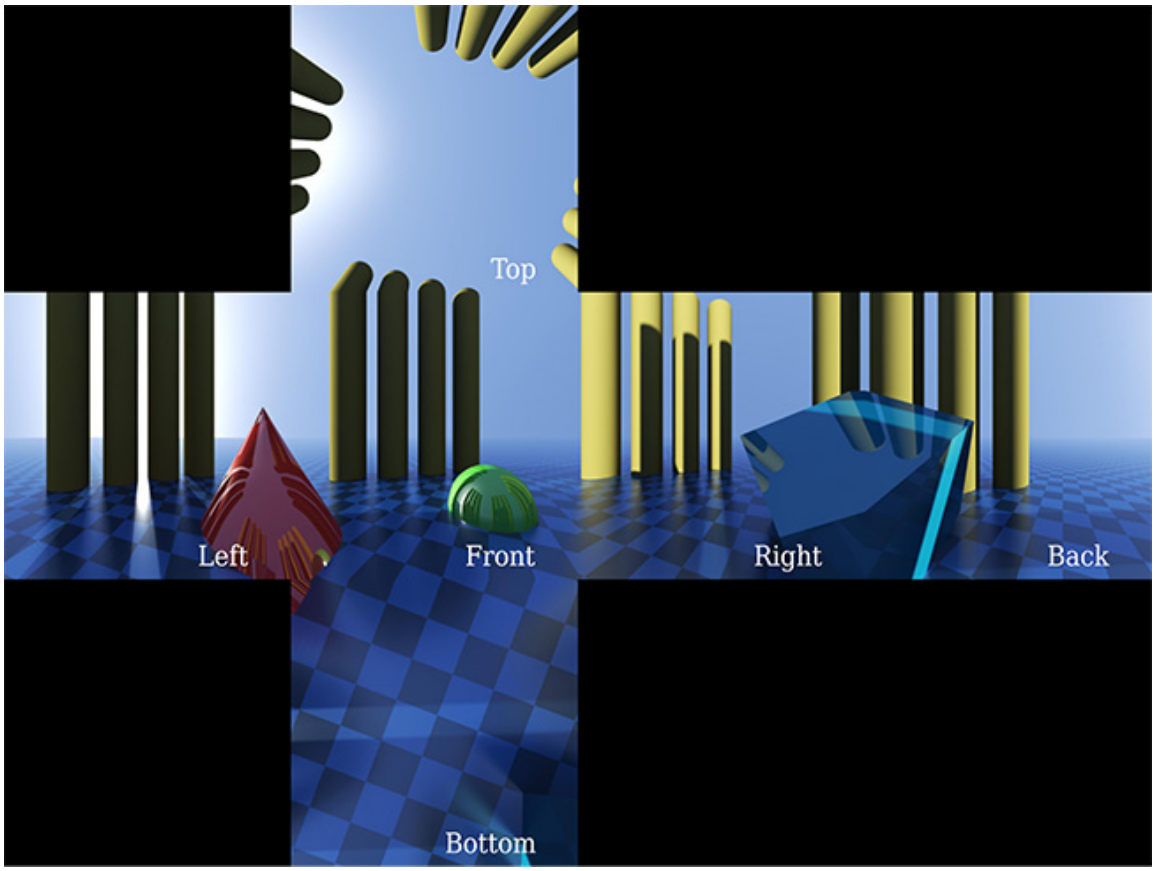
Easily include real-world video shot in 360 degrees.
Reduce **Scene** complexity in VR by including a pre-rendered backdrop video instead of real geometry.
Unity supports 180 and 360 degree videos in either an equirectangular layout (longitude and latitude) or a cubemap layout (6 frames).

Equirectangular 2D videos should have an **aspect ratio** of exactly 2:1 for 360 content, or 1:1 for 180 content.



Equirectangular 2D video

Cubemap 2D videos should have an aspect ratio of 1:6, 3:4, 4:3, or 6:1, depending on face layout:



Cubemap 2D video

To use the panoramic video features in the Unity Editor, you must have access to panoramic video clips, or know how to author them.

This page describes the following steps to display any panoramic video in the Editor:

Set up a Video Player to play the video source to a Render Texture.

Set up a Skybox Material that receives the Render Texture.

Set the Scene to use the Skybox Material.

Note: This is a resource-intensive feature. For best visual results, use panoramic videos in the highest possible resolution (often 4K or 8K). Large videos require more computing power and resources for decoding. Most systems have specific limits on maximum video decoding resolutions (for example, many mobiles are limited to HD or 2K, and older desktops might be limited to 2K or 4K).

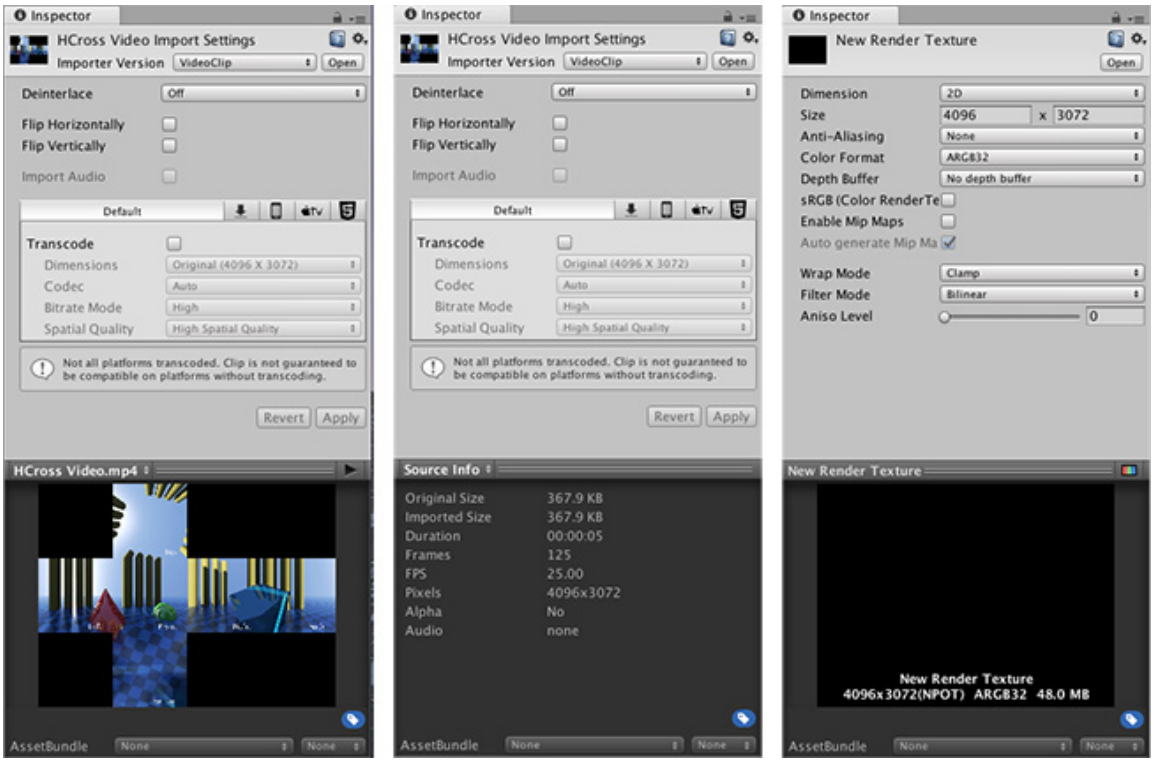
1. Video player setup

Import your video into Unity as an Asset. To create a Video Player, drag the video Asset from the **Project view** to an empty area of Unity's Hierarchy view. By default, this sets up the **component** to play the video full-screen for the default **Camera**. Press **Play** to view this.

You should change this behaviour so that it renders to a Render Texture. That way, you can control exactly how it is displayed. To do this, go to **Assets** > **Create** > **Render Texture**.

Set the Render Texture's **Size** to match your video exactly. To check the dimensions of your video, select the video in your Assets folder and view the **Inspector** window. Scroll to the section where Unity previews your video, select your video's name in the preview window, and change it to **Source Info**.

Next, set your Render Texture's **Depth Buffer** option to **No depth buffer**.

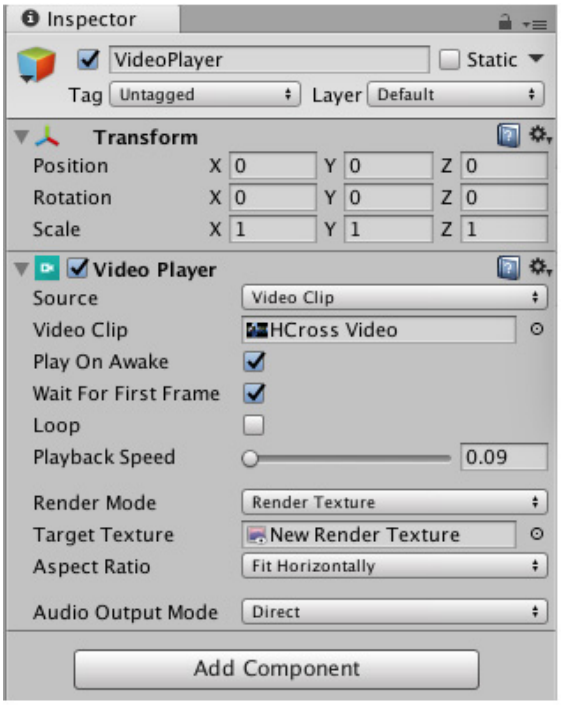


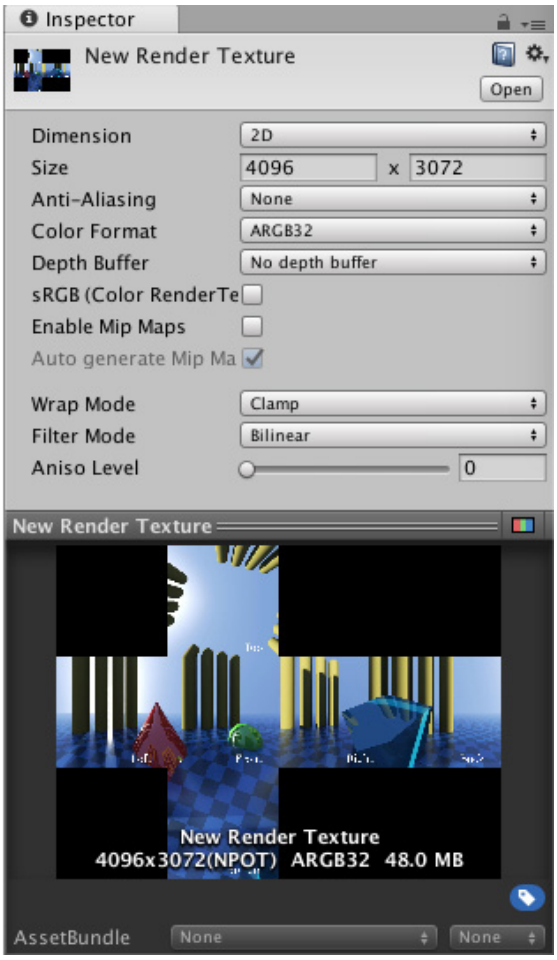
Render Texture set to **No depth buffer**

Now, open the **Video Player** Inspector and switch the **Render Mode** to **Render Texture**. Drag your new Render Texture from the Asset view to the **Target Texture** slot.

Enter Play mode to verify that this is functioning correctly.

The video doesn't render in the **Game** window, but you can select the Render Texture Asset to see that its content updating with each video frame.





2. Create a Skybox Material

You need to replace the default Skybox with your video content to render the panoramic video as a backdrop to your Scene.

To do this, create a new Material (**Assets** > **Create** > **Material**). Go to your new Material's Inspector and set the Material's Shader to Skybox/Panoramic (go to **Shader** > **Skybox** > **Panoramic**).

Drag the Render Texture from the Asset view to the Texture slot in the new Material's Inspector.

You must correctly identify the type of content in the video (cubemap or equirectangular) for the panoramic video to display properly. For cubemap content (such as a cross and strip layout, as is common for static Skybox Textures), set **Mapping** to **6 Frames Layout**.

For equirectangular content, set **Mapping** to **Latitude Longitude Layout**, and then either the **360 degree** or **180 degree** sub-option. Choose the **360 degree** option if the video covers a full 360-degree view. Choose **180 degree** if the video is just a front-facing 180-degree view.

Look at the **Preview** at the bottom of the Material Inspector. Pan around and check that the content looks correct.

3. Render the panoramic video to the Skybox

Finally, you must connect your new Skybox Material to the Scene.

Open up the Lighting window (menu: **Window** > **General** > **Lighting Settings**).

Drag and drop the new Skybox Material Asset to the first slot under **Environment**.

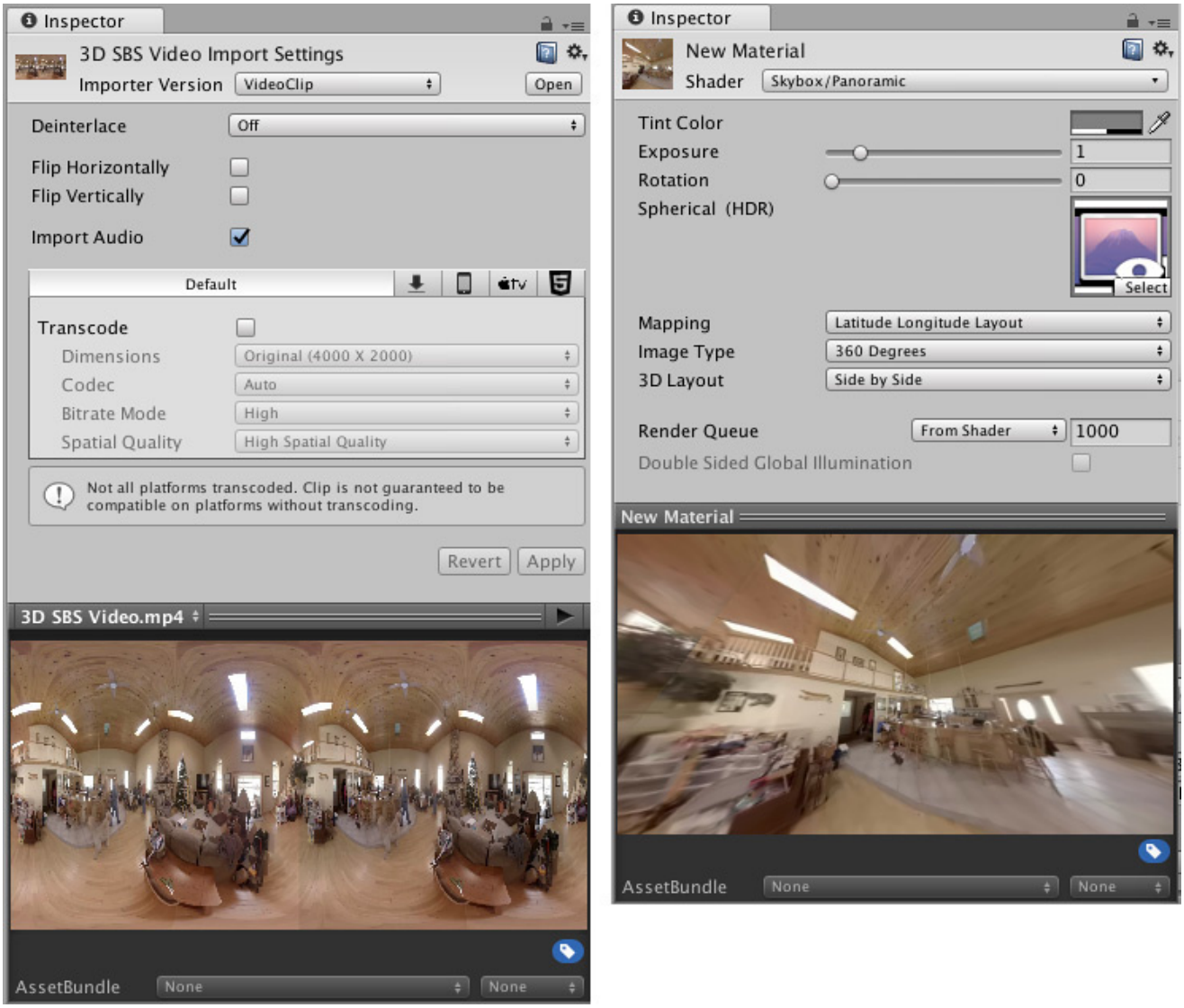
Press Play to show the video as a Scene backdrop on the Skybox.

Change the Scene Camera orientation to show a different portion of the Skybox (and therefore a different portion of the panoramic video).

3D panoramic video

Turn on Virtual Reality Support in the Player Settings (menu: **Edit** > **Project Settings** > **Player** >**XR Settings**), especially if your source video has stereo content. This unlocks an extra **3D Layout** option in the Skybox/Panoramic Material. The 3D Layout has three options : **Side by Side**, **Over Under**, and **None** (default value).

Use the **Side by Side** settings if the video contains the left eye's content on the left and the right eye's content on the right. Choose **Over Under** if the left and right content are positioned above and below one another in the video. Unity detects which eye is currently being rendered and sends this information to the Skybox/Panoramic shader using Single Pass Stereo rendering. The shader contains the logic to select the correct half of the video based on this information when Unity renders each eye's content in VR.



3D panoramic video

For non–360 3D video (where you shouldn't use a Skybox Material), the same **3D Layout** is available directly in the Video Player component when using the Camera **Near/Far** Plane Render Modes.

Alternate Render Texture type for cubemap videos

When working with cubemap videos, instead of the Video Player **rendering** to a 2D Render Texture and preserving the exact cube map layout, you can render the Video Player directly to a Render Texture Cube.

To achieve this, change the Render Texture Asset's **Dimension** from **2D** to **Cube and set the Render Texture's** Size__ to be exactly the dimensions of the individual faces of the source video.

For example, if you have a 4 x 3 horizontal cross cubemap layout video with dimensions 4096 x 3072, set the Render Texture's **Size** to 1024 x 1024 (4096 / 4 and 3072 / 3).

While rendering to a Cube Target Texture, the Video Player assumes that the source video contains a cube map in either a cross or a strip layout (which it determines using the video aspect ratio). The Video Player then fills out the Render Texture's faces with the correct cube faces.

Use the resulting Render Texture Cube as a Skybox. To do this, create a Material and assign **Skybox/Cubemap** as the **Shader**(**Shader** > **Skybox** > **Cubemap**) instead of the Skybox/Panoramic Material described above.

Video dimensions and transcoding

Including 3D content requires double either the width or the height of the video (corresponding to **Side-by-Side** or **Over-Under** layouts).

Keep in mind that many desktop hardware video decoders are limited to 4K resolutions and mobile hardware video decoders are often limited to 2K or less which limits the resolution of playback in real-time on those platforms.

You can use video transcoding to produce lower resolution versions of panoramic videos, but take precautions to avoid introducing bleeding at the edge between left and right 3D content, or, between cube map faces and adjacent black areas. In general, you should author video in power-of-two dimensions and transcoding to other power-of-two dimensions to reduce visual artifacts.

2017–10–25 Page published with limited editorial review

Added 2D and 3D panoramic video support in 2017.3

Terrain Engine

Leave feedback

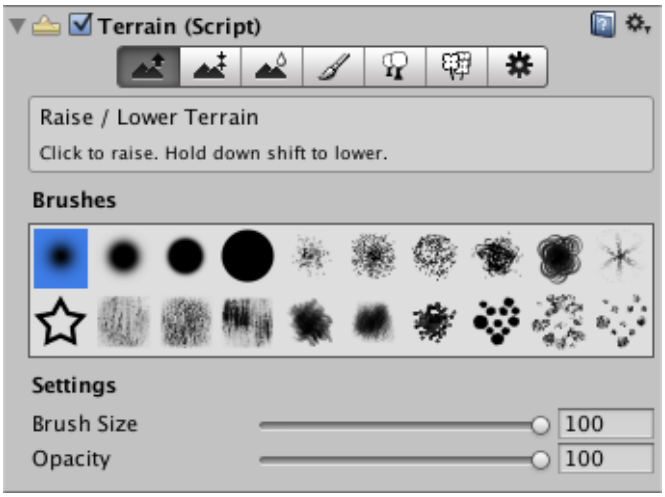
Unity's **Terrain** system allows you to add vast landscapes to your games. At runtime, terrain **rendering** is highly optimized for **rendering** efficiency while in the editor, a selection of tools is available to make terrains easy and quick to create. This section explains the various options available for terrains and how to make use of them.

See also the Knowledge Base Terrain section.

Creating and editing Terrains

Leave feedback

To add a Terrain GameObject to your Scene, select **GameObject** > **3D Object** > **Terrain** from the menu. This also adds a corresponding Terrain Asset to the Project view. When you do this, the landscape is initially a large, flat plane. The Terrain's Inspector window provides a number of tools you can use to create detailed landscape features.



Terrain Editing Tools appear in the Inspector

With the exception of the tree placement tool and the settings panel, all the tools on the **toolbar** provide a set of "brushes" and settings for brush size and opacity. It is no coincidence that these resemble the painting tools from an image editor because detail in the terrain is created precisely that way, by "painting" detail onto the landscape. If you select the leftmost tool on the bar (*Raise/Lower Terrain*) and move the mouse over the terrain in the **scene view**, you will see a cursor resembling a spotlight on the surface. When you click the mouse, you can paint gradual changes in height onto the landscape at the mouse's position. By selecting different options from the *Brushes* toolbar, you can paint with different shapes. The *Brush Size* and *Opacity* options vary the area of the brush and the strength of its effect respectively.

The details of all of the tools will be given in subsequent sections. However, all of them are based around the concept of painting detail and with the exception of the tree placement tool, all have the same options for brushes, brush size and opacity. The tools can be used to set the height of the terrain and also to add coloration, plants and other objects.

Terrain Keyboard Shortcuts

You can use the following keyboard shortcuts in the terrain **inspector**:

Press the keys from F1 to F6 to select the corresponding terrain tool (for example, F1 selects the *Raise/Lower* tool).
The comma (,) and period (.) keys cycle through the available brushes.
Shift-comma (<) and Shift-dot (>) cycle through the available objects for trees, Textures and details.

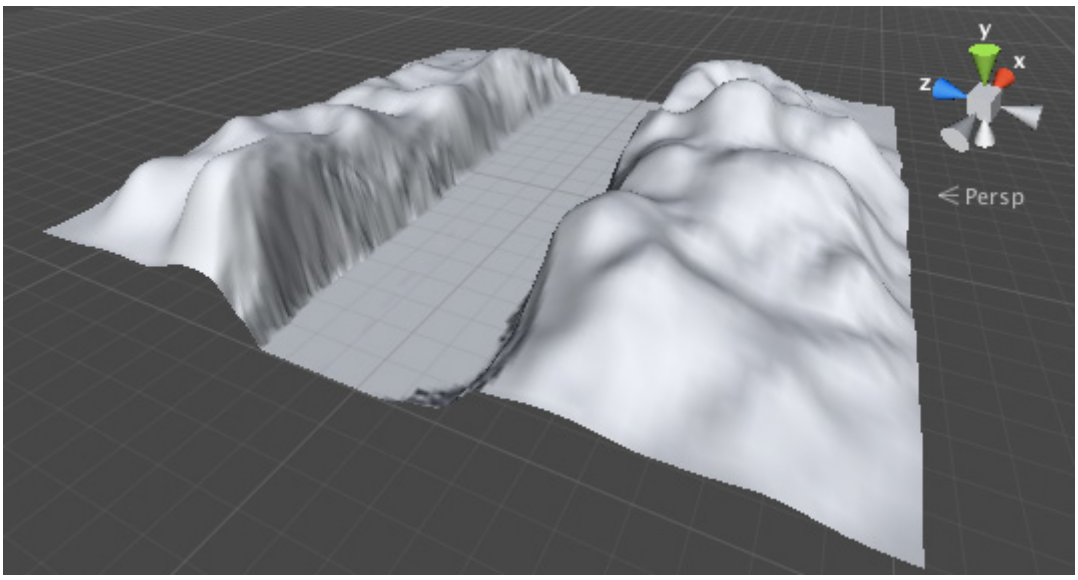
Additionally, the standard F keystroke works slightly differently for terrains. Normally, it frames the selection around the whole object when the mouse is over the **scene** view. However, since terrains are typically very large, pressing F will focus the **scene** view on the area of terrain where the mouse/brush is currently hovering. This provides a very quick and intuitive way to jump to the area of terrain you want to edit. If you press F when the mouse is away from the terrain object, the standard framing behaviour will return.

Height Tools

Leave feedback

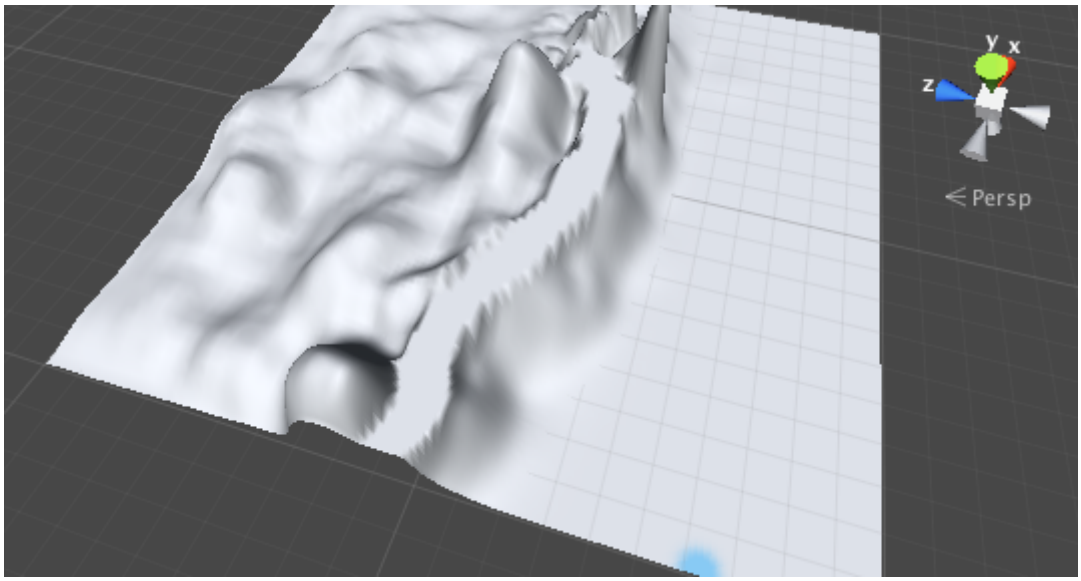
The first three tools on the **terrain inspector toolbar** are used to paint changes in height onto the **terrain**.

From the left, the first button activates the *Raise/Lower Height* tool. When you paint with this tool, the height will be increased as you sweep the mouse over the terrain. The height will accumulate if you hold the mouse in one place, similar to the effect of the airbrush tools in image editors. If you hold down the shift key, the height will be lowered. The different brushes can be used to create a variety of effects. For example, you can create rolling hills by increasing the height with a soft-edged brush and then cut steep cliffs and valleys by lowering with a hard-edged brush.



Rolling hills cut by a steep valley

The second tool from the left, *Paint Height* is similar to the *Raise/Lower* tool except that it has an additional property to set the target height. When you paint on the object, the terrain will be lowered in areas above that height and raised in areas below it. You can use the *Height* property slider to set the height manually or you can shift-click on the terrain to sample the height at the mouse position (rather like the “eyedropper” tool in an image editor). Next to the *Height* property is a *Flatten* button that simply levels the whole terrain to the chosen height. This is useful to set a raised ground level, say if you want the landscape to include both hills above the level and valleys below it. *Paint Height* is handy for creating plateaux in a **scene** and also for adding artificial features like roads, platforms and steps.



Hillside with a flat road

The third tool from the left, *Smooth Height* does not significantly raise or lower the terrain height but rather averages out nearby areas. This softens the landscape and reduces the appearance of abrupt changes, somewhat like the blur tool in an image editor. You might use this, for example, when you have painted detail using one of the noisier brushes in the available set. These brush patterns will tend to introduce sharp, jagged rocks into a landscape, but these can be softened using *Smooth Height*.

Working with Heightmaps

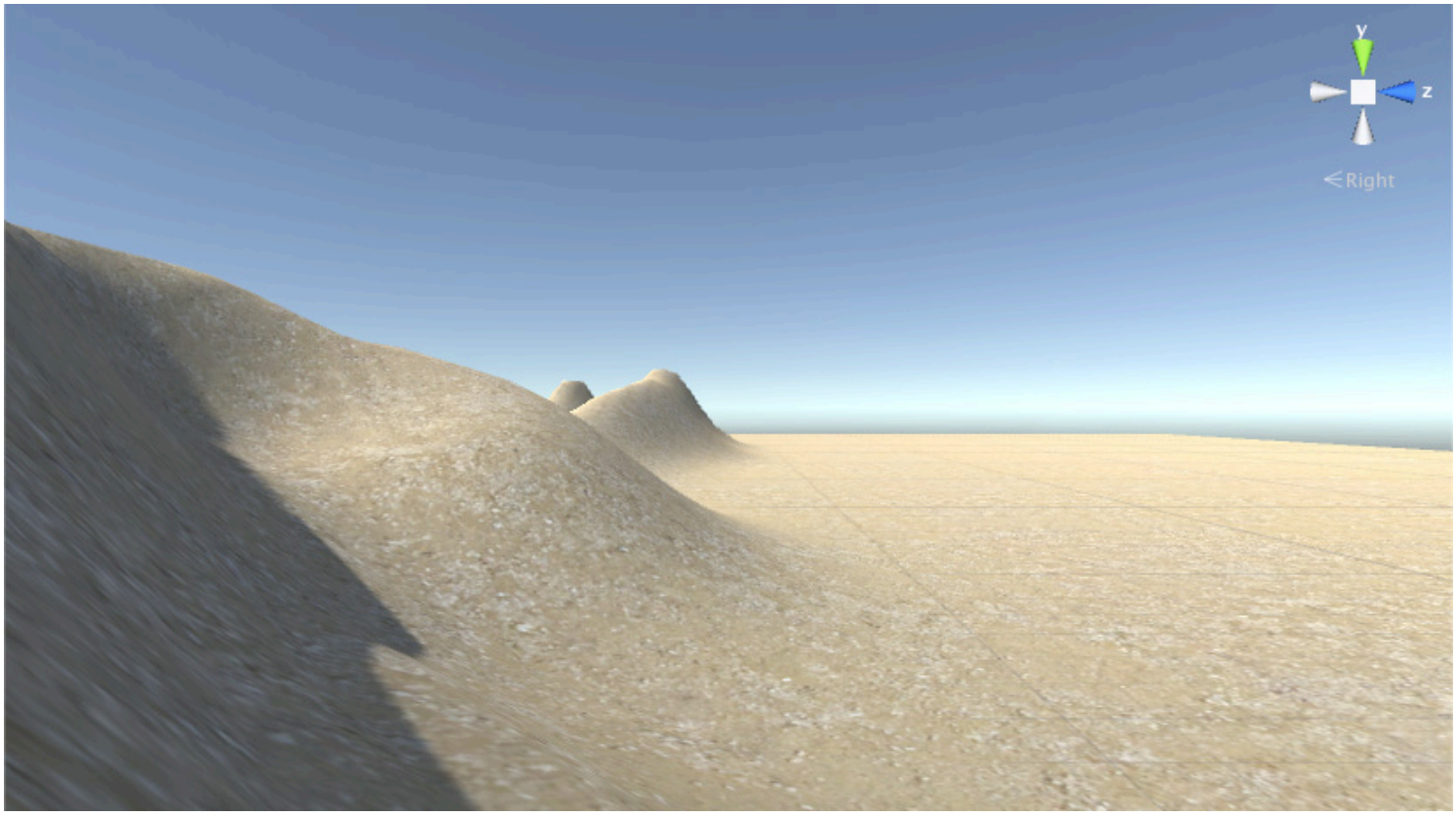
As noted above, the height tools are reminiscent of painting tools available in image editors. In fact, the terrain is implemented using a texture behind the scenes and so the tools are ultimately acting as texture painting tools.

The height of each point on the terrain is represented as a value in a rectangular array. This array can be represented using a grayscale image known as a **heightmap**. It is sometimes useful to work on a heightmap image in an external editor, such as Photoshop, or obtain existing geographical heightmaps for use in your game. Unity provides the option to import and export heightmaps for a terrain; if you click on the *Settings* tool (the rightmost button in the toolbar) you will find buttons labelled *Import RAW* and *Export RAW*. These allow the heightmap to be read from or written to the standard RAW format, which is a 16-bit grayscale format compatible with most image and landscape editors.

Terrain Textures

Leave feedback

You can add texture images to the surface of a terrain to create coloration and fine detail. Since **terrains** are such large objects, it is standard practice to use a texture that repeats seamlessly and tile it over the surface (the repeat generally isn't noticeable from a character's viewpoint close to the ground). One texture will serve as the "background" image over the landscape but you can also paint areas of different textures to simulate different ground surfaces such as grass, desert and snow. The painted textures can be applied with variable transparency so you can have a gradual transition between grassy countryside and a sandy beach, for example.



Sand dune terrain with sandy texture

Enabling Textures

The paintbrush button on the **toolbar** enables texture painting.

Initially, the terrain has no textures assigned for painting. If you click the *Edit Textures* button and select *Add Texture* from the menu, you will see the *Add Terrain Texture* window. Here you can set a texture and its properties.

Depending on the material type you set in Terrain Settings, the color channels of the main texture map may have different uses. These are listed below in *Terrain Texture Settings*.

Click on *Select* to see your texture **assets** in a separate *Select Texture* window (not shown). Click on the texture you want and it displays in the *Add Terrain Texture* window. (See *Add Terrain Texture* window, before and after, in *Fig 1* below.)

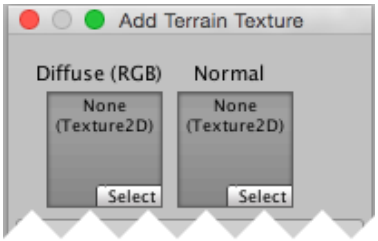
Fig 1: Click on *Select* in the Add Terrain Texture window and choose a texture asset from the Select Texture window (not shown) - it then displays, ready to add to the terrain

Terrain Texture Settings

Depending on the material type you set in Terrain Settings, the color channels of the main texture map may have different uses. The different *Add Terrain Texture* windows are listed here.

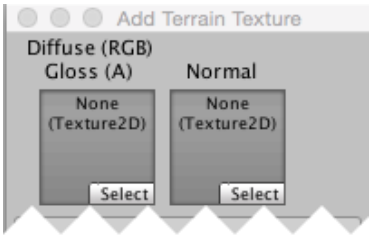
Standard [image above in *Fig 1*]: RGB channels are the albedo color of the terrain surface, while alpha channel controls the smoothness. There is also a 'Metallic' slider which controls the overall look of the surface.

Diffuse: RGB channels are the diffuse color. Alpha channel is not used.



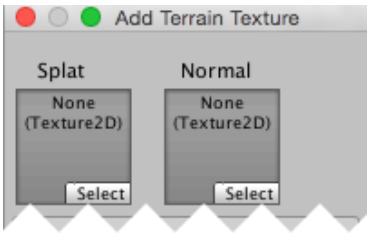
Add Texture window (Diffuse)

Specular: RGB channels are the diffuse color. Alpha channel is the gloss map.



Add Texture window (Specular)

Custom: How the splat map is used depends on your custom **shader**, but usually you want the RGB channels to be the base color.



Add Texture window (Custom)

Besides the main texture map, you can also specify a normal texture for all of the 3 built-in material types. The texture type of the normal texture used here must be 'Normal Map' (you can change the texture type of a texture asset in its import settings). Shader code that handles normal map will be turned on only when at least one normal texture is set for the terrain, so you don't need to pay the performance cost of normal map if you don't use them.

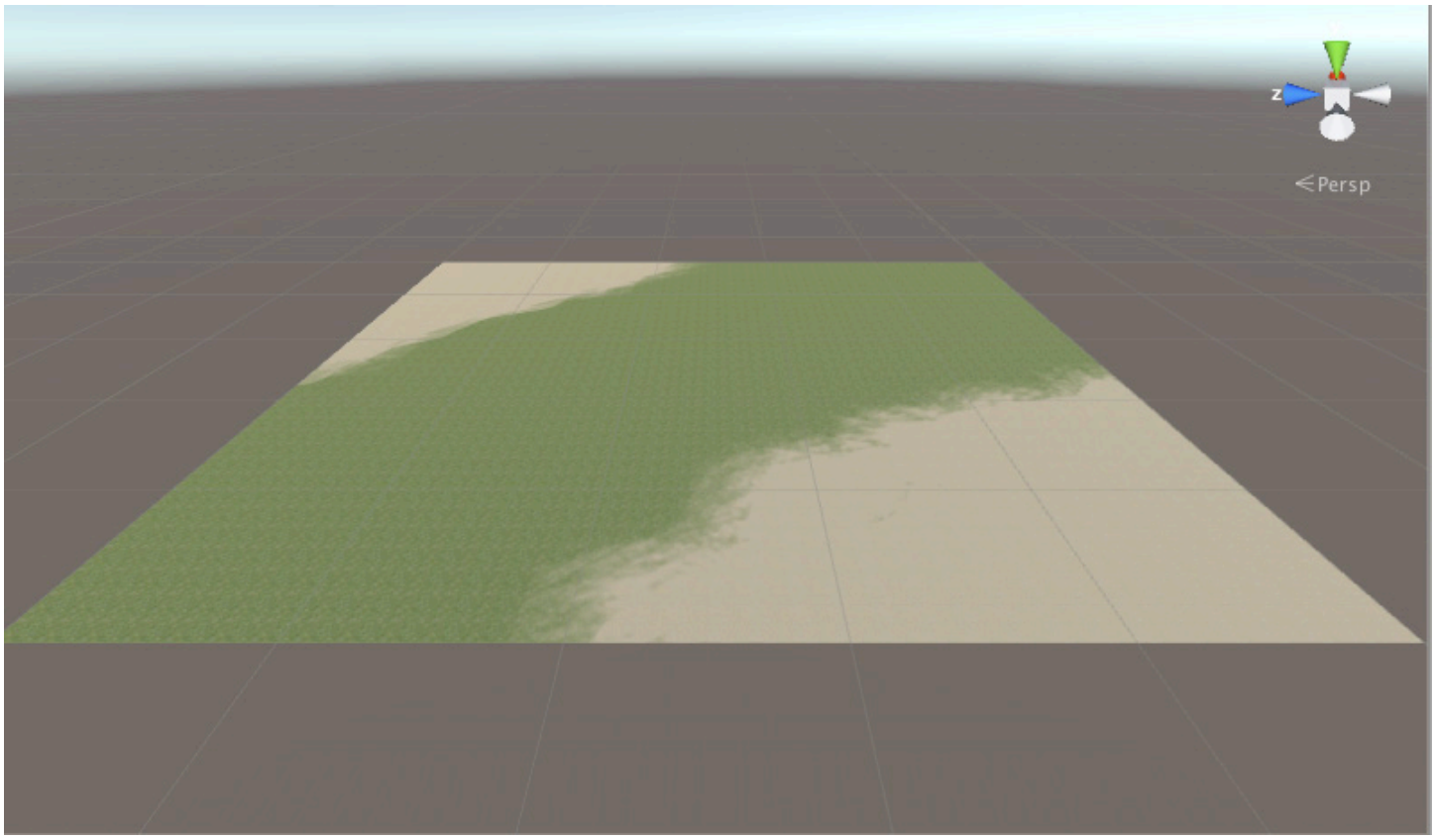
The *Size* property (just below the texture boxes) lets you set the width and height over which the image will stretch on the terrain's surface. The *Offset* property determines how far from the terrain's anchor point the tiling will start; you can set it to zero to start the tiling right in the corner. Once you have set the texture and properties to your liking, click the **_Apply** button to make the texture available to the terrain.

To make changes to an added terrain texture, select its thumbnail, click the 'Edit Textures' button and select 'Edit Texture...' from the menu. Or, you can simply double click on its thumbnail. To remove a terrain texture, select its thumbnail, click the 'Edit Textures' button and select 'Remove Texture' from the menu.

Note that if you want to assign a Texture to a Terrain, you need to open the Texture Importer and tick the **Read/Write Enabled** checkbox.

Texture Painting

The first texture you add will be used as a "background" to cover the terrain. However, you can add as many textures as you like; the subsequent ones will be available for painting using the familiar brush tools. Below the textures in the Terrain **inspector**, you will see the usual *Brush Size* and *Opacity* options but also an additional option called *Target Strength*. This sets the maximum opacity value that the brush will build up even if it passes over the same point repeatedly. This can be a useful way to add subtle patches of color variation within a single terrain type to break the monotony of a large, flat area with the same texture tile repeating over and over.



Grass terrain with dirt texture painted on corners

Trees

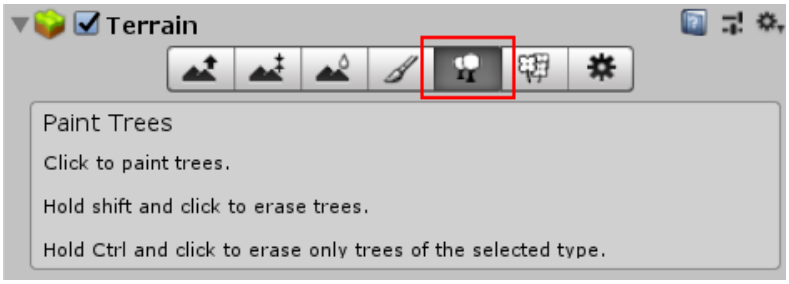
Leave feedback

To enhance Unity Terrains, you can paint Trees onto a Terrain in much the same way as painting height maps and textures. However, Trees are solid 3D GameObjects that grow from the surface. Unity uses optimisations, like **billboarding** for distant Trees, to maintain good **rendering** performance. This means that you can have dense forests with thousands of trees and still keep an acceptable frame rate.

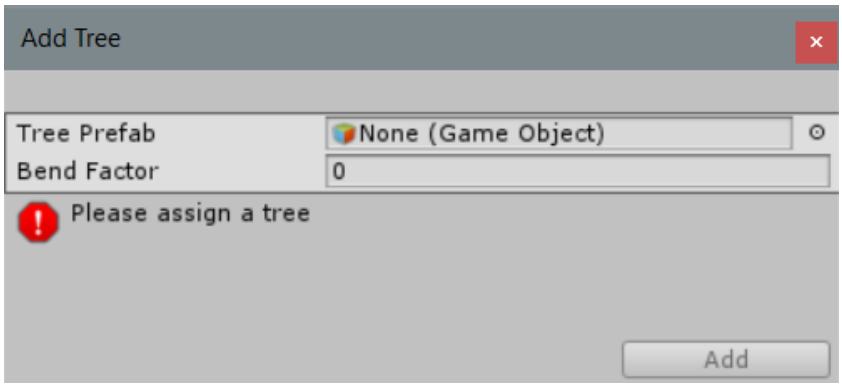
Terrain with Trees

Painting Trees

The Tree button on the **toolbar** enables Tree painting:



Initially, the Terrain will have no Trees available. In order to start painting onto the Terrain, you need to add a Tree. Click the **Edit Trees** button and select **Add Tree**. From here, you can select a Tree **asset** from your Project and add it as a **Tree Prefab** for use with the brush:



The Add Tree window

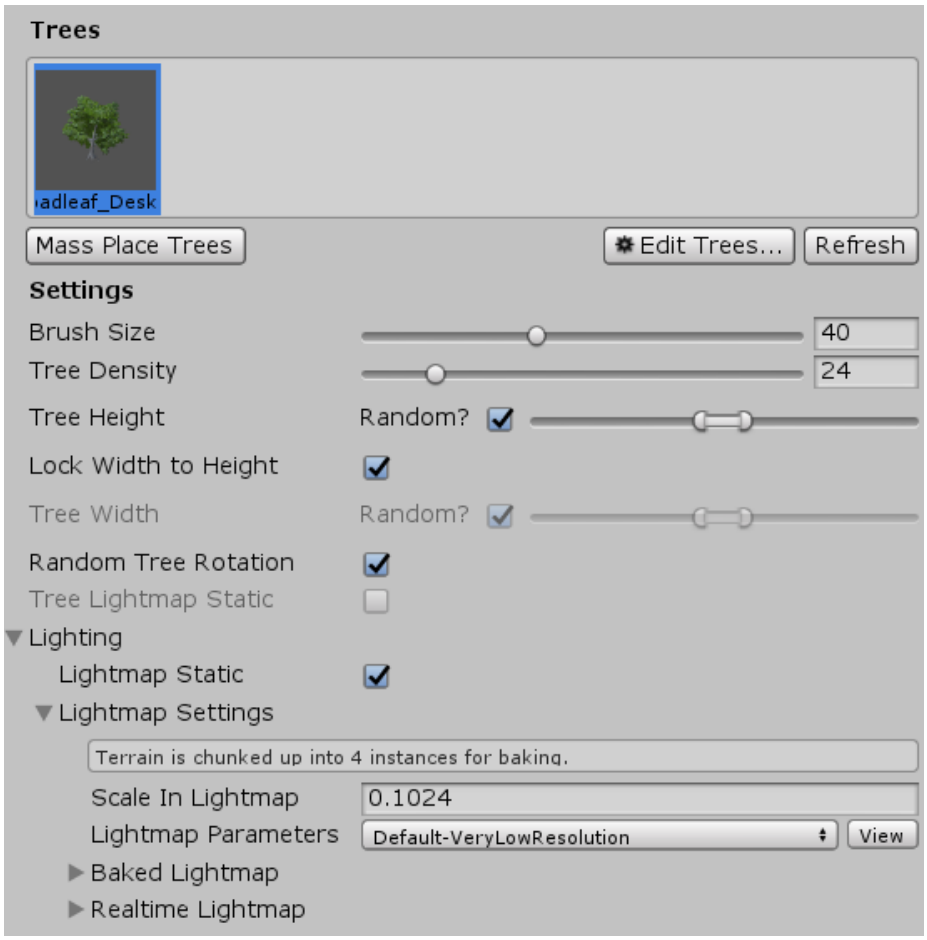
To help prototyping, Unity provides several sample SpeedTree Tree GameObjects in the **Standard Assets** package. Alternatively, you can create your own Trees.

If the Tree GameObject that you are importing supports **Bend Factor** , the Add Tree window shows a **Bend Factor** property for adjusting wind responsiveness. Trees created using the SpeedTree Modeller have a **Bend Factor**. See the section on **Making Trees bend in the wind** below.

When you have set up your Tree properties (described below), you can paint Trees onto the Terrain in the same way you paint textures or heightmaps. You can remove Trees from an area by holding the shift key while you paint, or remove just the currently selected Tree type by holding down the control key.

Tree properties

After you have selected which Tree to place, adjust its settings to customise Tree placement and characteristics:



Tree properties

Property:	Function:

Property:	Function:
Mass Place Trees	Create an overall covering of Trees without painting over the whole landscape. After mass placement, you can still use painting to add or remove Trees to create denser or sparser areas.
Brush Size	Controls the size of the area that you can add Trees to.
Tree Density	Tree Density controls the average number of Trees painted onto the area defined by Brush Size.
Tree Height	Control the Tree's minimal height and maximal height using a slider. Drag the slider to the left for short Trees, and right for tall Trees. If you uncheck Random, you can specify the exact scale for the height of all newly painted Trees within the range of 0.01 to 2.
Lock Width to Height	By default, a Tree's width is locked to its height so that Trees are always scaled uniformly. However, you can disable the Lock Width to Height option and specify the width separately.
Tree Width	If the Tree's width is not locked to its height you can control the Tree's minimal width and maximal width using a slider. Drag the slider to the left for thin Trees, and right for wide Trees. If you uncheck Random, you can specify the exact scale for the width of all newly painted Trees within the range of 0.01 to 2.
Random Tree Rotation	Random Tree Rotation is a variation option used to help create the impression of a random, natural-looking forest rather than an artificial plantation of identical Trees. Untick this if you want to place Trees with fixed, identical rotations.
Lighting	
Lightmap Static	Enable this check box to indicate to Unity that the GameObject's location is fixed and it will participate in Global Illumination computations. If a GameObject is not marked as **Lightmap** Static then it can still be lit using **Light Probes**.
Scale In Lightmap	Specifies the relative size of the GameObject' UVs within a lightmap. A value of 0 will result in Unity not lightmapping the GameObject, but the GameObject will still contribute to the lighting of other GameObjects in the **Scene**. A value greater than 1.0 increases the number of **pixels** in the lightmap used for this GameObject. A value less than 1.0 decreases the number of **pixels**. You can use this property to optimise lightmaps so that important or detailed areas are more accurately lit. For example, an isolated building with flat, dark walls will use a low lightmap scale (less than 1.0), while a collection of colorful motorcycles displayed close together need a high scale value.
Lightmap Parameters	Allows you to choose or create a set of Lightmap Parameters for this GameObject.

Creating Trees

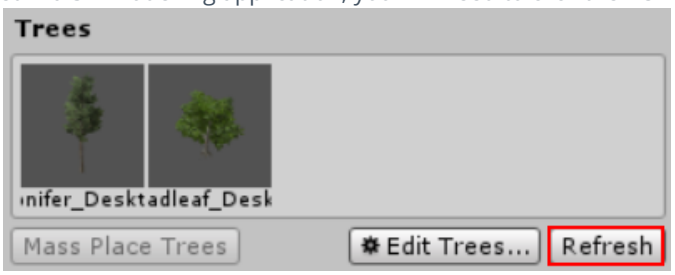
If you want to create your own Trees, you can use the SpeedTree Modeler, Unity's Tree Creator tool, or any 3D modelling application, and then import them into your Project.

You can use SpeedTree Modeler (from IDV, Inc.) to create Trees with advanced visual effects such as smooth **LOD** transition, fast billboarding and natural wind animation. For more detailed information, refer to the SpeedTree Modeler documentation. You can also import SpeedTree assets into your Project folder from **Asset Store** packages or other third party sources.

Unity has its own Tree creator that you can use to produce new Tree assets. You can also use a 3D modelling application.

When creating Trees, position the anchor point at the base of the Tree where it emerges from the ground. For performance reasons, your Tree **mesh** should have fewer than 2000 triangles. and The **mesh** always has exactly two materials: one for the Tree body and the other for the leaves.

Trees must use the **Nature/Soft Occlusion Leaves** and **Nature/Soft Occlusion Bark shader**. To use those shaders, you have to place Trees in a specific folder named **Ambient-Occlusion**, or the Trees won't render correctly. When you place a model in such a folder and reimport it, Unity will calculate soft **ambient occlusion** in a way that is specifically designed for Trees.

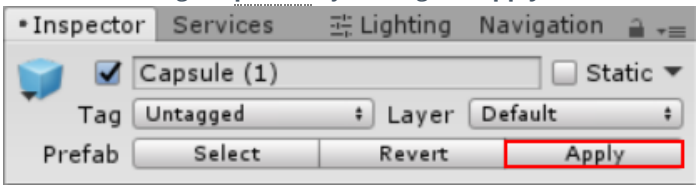
If you change an imported Tree asset in a 3D modelling application, you will need to click the **Refresh** button in the Editor in order to see the updated Trees on your Terrain: 

Warning When importing and altering a SpeedTree model in a 3D modeling program, when you re-export it (as an .fbx or .obj) you may lose the natural wind animation functionality that comes with SpeedTree models.

Using Colliders with Trees

You can add a Capsule Collider to a new Tree asset by instantiating the Tree in the scene by dragging the Prefab from your Assets folder into the Scene. Then add the collider using menu: **Component** > **Physics** > **Capsule Collider**. You can then **either**:

Override the original **prefab** by clicking the **Apply** button on the Tree GameObject in the Inspector Window:



Or, create a new Prefab by dragging the Tree GameObject into your Assets folder.

When you add the Tree to the Terrain for painting, make sure that if you have created a new Prefab that you select the correct one with with the **collider** rather than the original GameObject. You must also enable **Create Tree Colliders** in the Terrain's **Terrain Collider** component inspector if you want to make .

Making Trees bend in the wind

You first need to create a **Wind Zone** to make Trees react to the wind. To do this select **GameObject** > **3D Object** > **Wind Zone**.

At this point, make sure that your Trees are set to bend. Select your **Terrain**, click the **Place Trees** button in the **Inspector**, and then select **Edit Trees > Edit Tree**. Setting the **Bend Factor** to 1 will cause the Trees to adjust if you have not already done this.

With the default settings, your Trees will move quite violently. To fix this, change your bend value in each individual Tree type. This is useful if you want some Tree types to bend more than others. To change the bend effect in the entire **Wind Zone**, set the values in the **Wind Zone** component directly. To reduce the fluttering effect of the leaves, adjust the wind turbulence down to around 0.1 to 0.3, and everything will become much smoother. If you don't want the Trees blowing all the way to one side and instead want some variation, set the **Wind Main** value down to the same value as your turbulence.

Tree Level of Detail (LOD) transition zone

Unity's LOD system uses a 2D to 3D transition zone to blend 2D **billboards** with 3D tree models seamlessly. This prevents any sudden popping of 2D and 3D trees, which is vital in VR. For more information about configuring LOD components, see the (LOD)[LevelOfDetail] and (LOD Group)[class-LODGroup] manual pages.

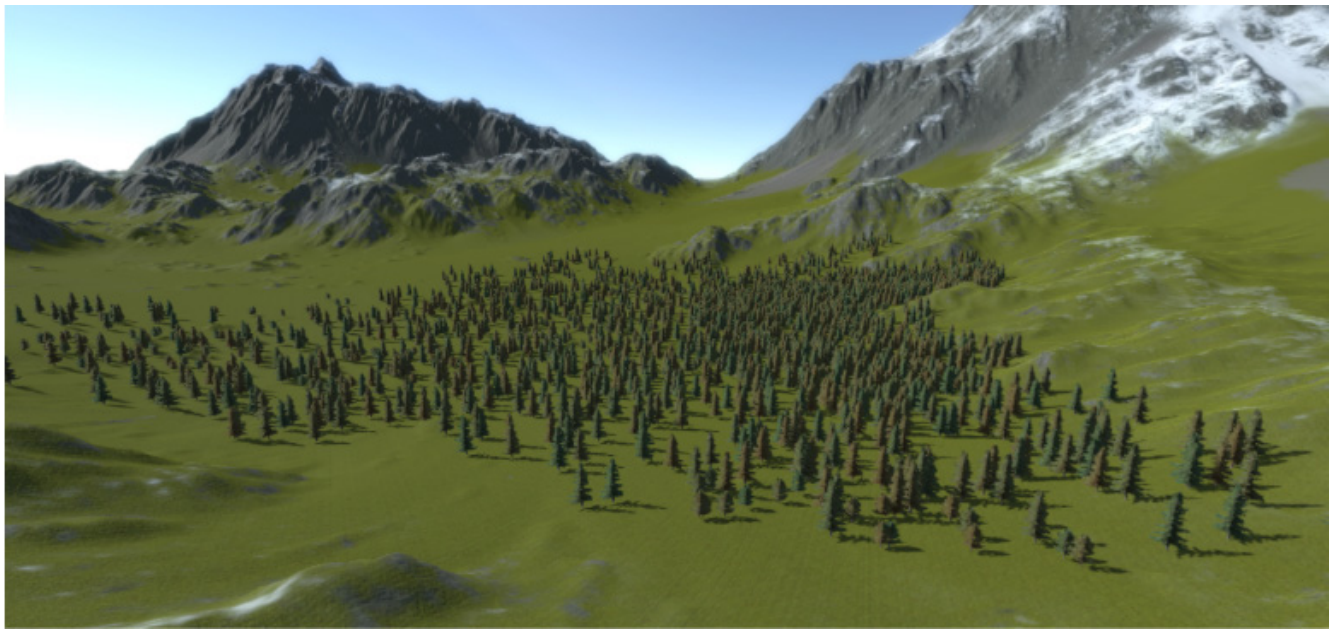
2018–09–20 Page amended with editorial review

GameObject menu changed in Unity 4.6

2D to 3D transition zone added to LOD system in Unity 2017.3

SpeedTree

Leave feedback



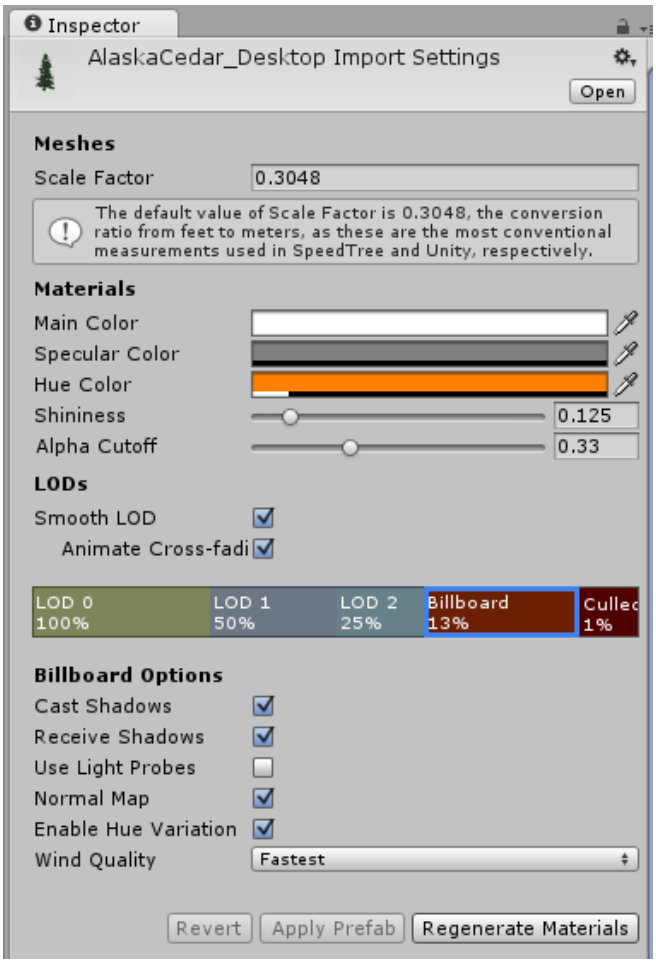
SpeedTree **Assets** (.spm files saved by Unity version of SpeedTree Modeler) are recognized and imported by Unity just like other Assets. Ensure Textures are reachable in the Project folder, and Materials for each LOD are generated automatically. There are import settings when you select .spm Assets for you to tweak the generated **GameObject** and attached Materials. Materials are not generated again on reimporting, unless you hit the **Generate Materials**, or **Apply & Generate Materials** button. Therefore any customisations to the Materials can be preserved.

The SpeedTree importer in the end generates a **Prefab** with LODGroup component configured. The Prefab can both be instantiated in a **Scene** as a common Prefab instance, or be selected as a tree prototype on the **terrain** and be "painted" across it. Additionally, **terrain** accepts any GameObject with an LODGroup **component** as a tree prototype, and put no limitation on the **mesh** size or number of Materials being used (in contrast to the Tree Creator trees). But, be aware that SpeedTree trees usually use 3–4 different Materials, which results in a number of draw-calls being issued every frame, so you should try to avoid heavy use of LOD trees on platforms that are sensitive to draw-call numbers.

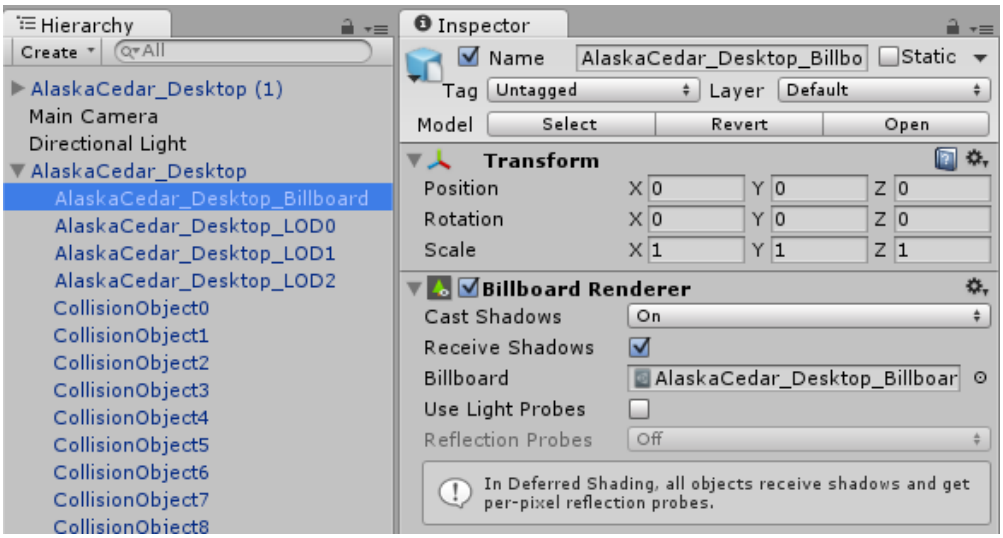
Casting and receiving real-time shadows

To make **billboards** cast shadows correctly, during the shadow caster pass, **billboards** are rotated to face the light direction (or light position in the case of point light) instead of facing the **camera**.

To enable these options, select the **Billboard LOD** level in the **Inspector** of an .spm Asset, tick **Cast Shadows** or **Receive Shadows** in **Billboard Options**, and click **Apply Prefab**.



To change billboard shadow options of instantiated SpeedTree GameObjects, select the billboard GameObject in the Hierarchy window, and tweak these options in the Inspector of the **Billboard Renderer**, just as you would with a normal **Mesh Renderer**.



Trees painted on the terrain inherit billboard shadow options from the Prefab.

You can use `BillboardRenderer.shadowCastingMode` and `BillboardRenderer.receiveShadows` to alter these options at runtime.

Known Issues: As with any other renderer, the **Receive Shadows** option has no effect while using deferred **rendering**. Billboards always receive shadows in deferred path.

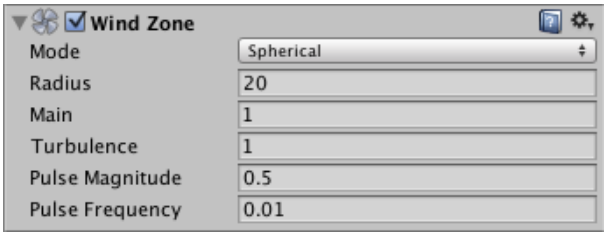
Wind Zones

Leave feedback

You can create the effect of wind on your **terrain** by adding one or more objects with **Wind Zone components**. Trees within a wind zone will bend in a realistic animated fashion and the wind itself will move in pulses to create natural patterns of movement among the tree.

Using Wind Zones

A wind zone object can be created directly (menu: **GameObject > 3D Object > Wind Zone**) or you can add the component to any suitable object already in the scene (menu: **Component > Miscellaneous > Wind Zone**). The inspector for the wind zone has a number of options to control its behaviour.



Wind Zone inspector

The *Mode* can be set to *Directional* or *Spherical*. In *Directional* mode, the wind will affect the whole terrain at once while a *Spherical* wind blows outwards within a sphere defined by the *Radius* property. Directional winds are more useful for creating natural movement of the trees while spherical winds are more suitable for special effects like explosions.

The *Main* property determines the overall strength of the wind but this can be given a little random variation using *Turbulence*. As mentioned above, the wind blows over the trees in pulses to create a more natural effect. The strength of the pulses and the time interval between them can be controlled using the *Pulse Magnitude* and *Pulse Frequency* properties.

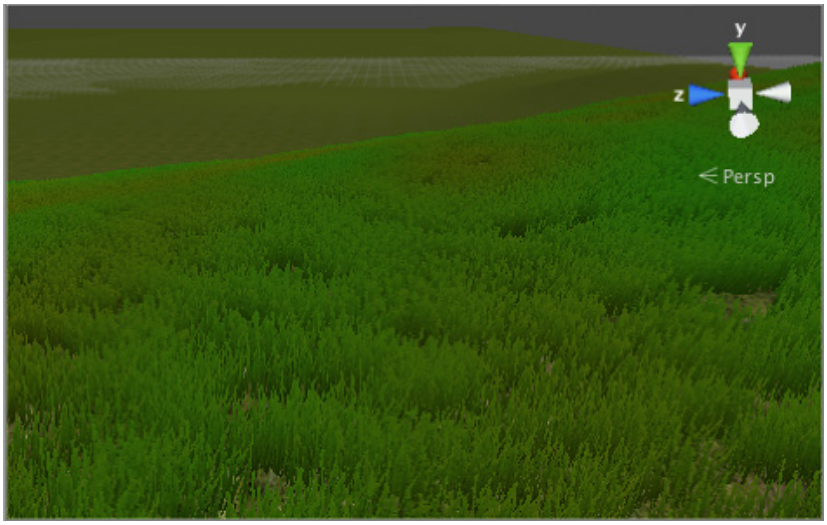
Particle Systems

The main use of wind is to animate trees but it can also affect particles generated by a particle system using the *External Forces* module. See the reference page for Particle Systems for further details.

Grass and other details

Leave feedback

A **terrain** can have grass clumps and other small objects such as rocks covering its surface. Grass is rendered by using 2D images to represent the individual clumps while other details are generated from standard meshes.



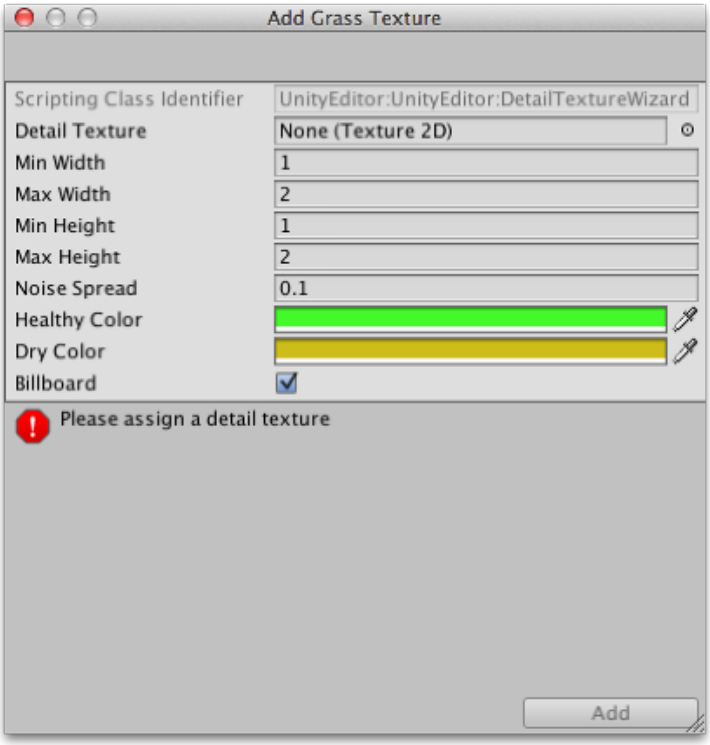
Terrain with grass

Enabling details

The details button on the **toolbar** enables grass/detail painting.

Initially, the terrain has no grass or details available but if you click the *Edit Details* button in the **inspector**, you will see the *Add Grass Texture* and *Add Detail Mesh* options on the menu that appears. A window will appear to let you choose the **assets** you want to add to the terrain for painting.

For grass, the window looks like this:



The Add Grass Texture window

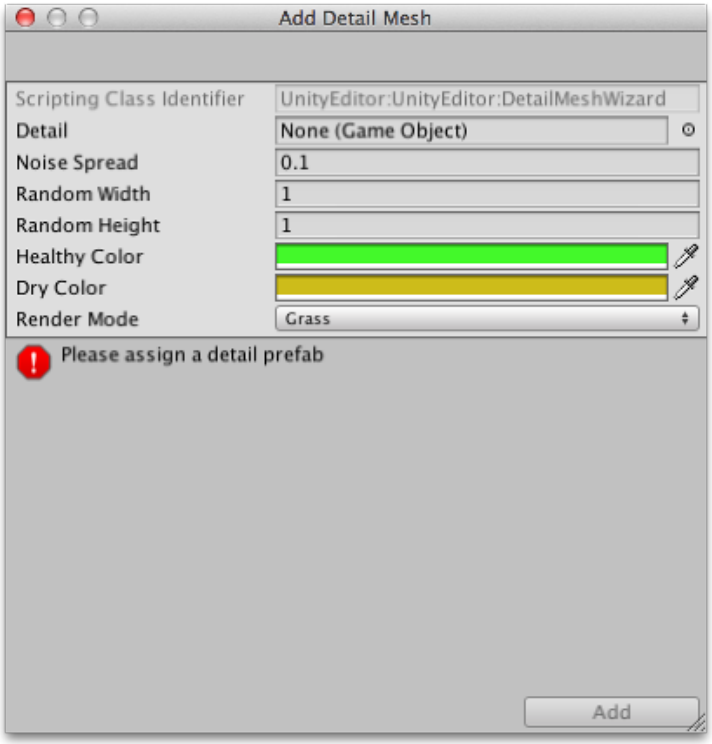
The *Detail Texture* is the texture that represents the grass. A few suitable textures are included in the Unity **Standard Assets** downloadable from the **Asset Store**. You can also create your own. The texture is simply a small image with alpha set to zero for the empty areas. ("Grass" is a generic term, of course - you can use the images to represent flowers, brush and perhaps even artificial objects like barbed wire coils.)

The *Min Width*, *Min Height*, *Max Width* and *Max Height* values specify the upper and lower limits of the size of the clumps of grass that are generated. For an authentic look, the grass is generated in random "noisy" patterns that have bare patches interspersed with the grass.

The *Noise Spread* value controls the approximate size of the alternating patches, with higher values indicating more variation within a given area. (Tech note: the noise is actually generated using *Perlin noise*; the noise spread refers to the scaling applied between the x,y position on the terrain and the noise image.) The alternating patches of grass are considered more "healthy" at the centres than at the edges and the *Healthy/Dry Color* settings show the health of grass clumps by their color.

Finally, when the *Billboard* option is enabled, the grass images will rotate so that they always face the **camera**. This option can be useful when you want to show a dense field of grass because there is no possibility of seeing clumps side-on and therefore visibly two-dimensional. However, with sparse grass, the rotations of individual clumps can become apparent, creating a strange effect.

For detail meshes, such as rocks, the selection window looks like this:-



The Add Detail Mesh window

The *Detail* property is used to select a **prefab** from your project which will be scaled by the *Random Width* and *Random Height* values for individual instances. The *Noise Spread* and *Healthy/Dry Color* values work the same as they do for grass (although the concept of "healthy" is stretched somewhat when applied to objects like rocks!) The *Render Mode* can be set to *Grass* or *Vertex Lit*. In *Grass Mode*, the instances of detail objects in the **scene** will be flattened into 2D images that behave like the grass textures. In *Vertex Lit* mode, the details will be rendered as solid, vertex lit objects in the **scene**.

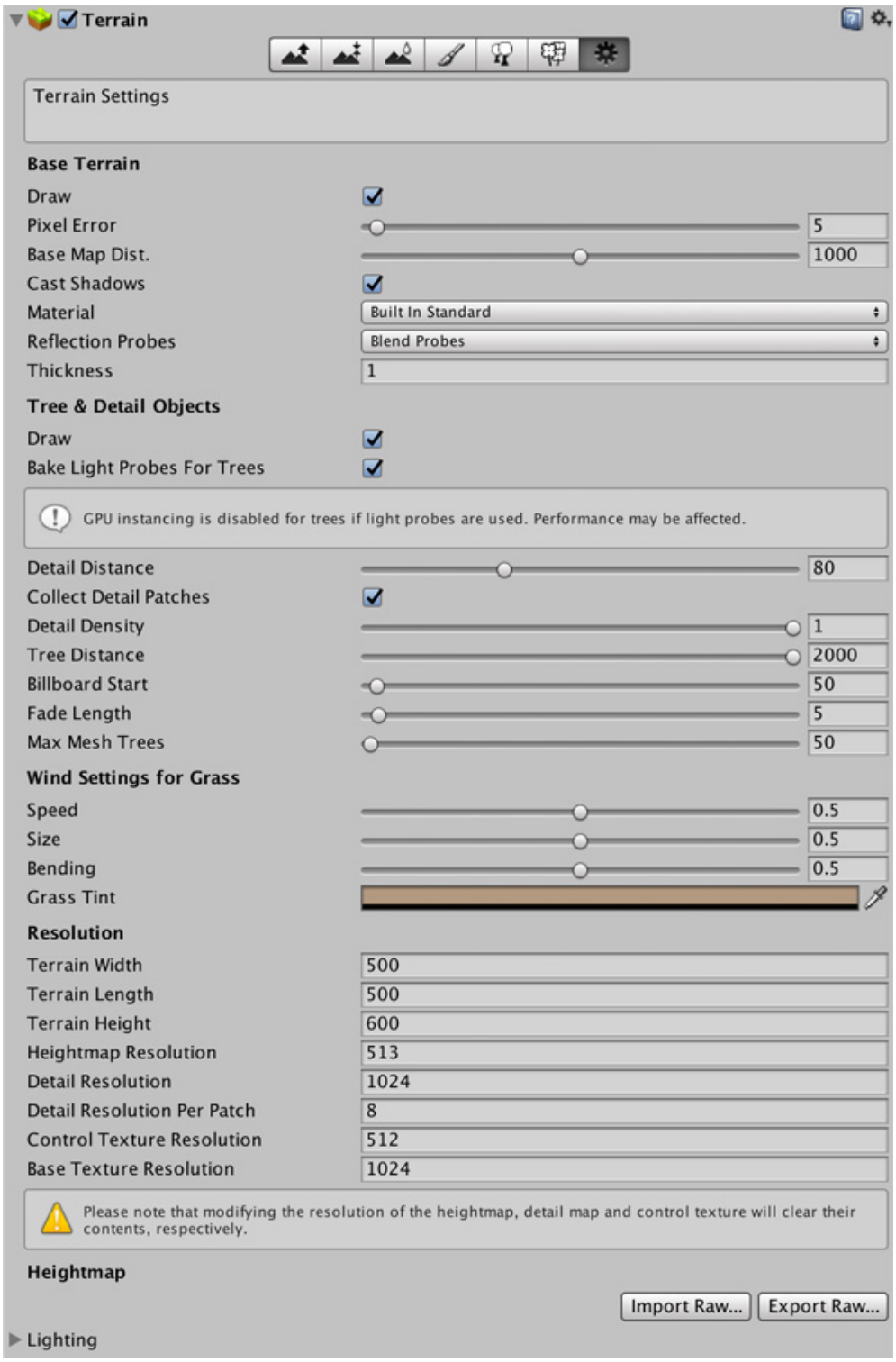
Terrain settings

Leave feedback

The final tool on the **terrain toolbar** is for settings:

Settings Inspector

Settings are provided for a number of overall usage and **rendering** options as described below:



Base Terrain

Property:	*Function:*
Draw	Toggle the rendering of terrain on / off.

Property:	*Function:*
Pixel Error	The accuracy of the mapping between the terrain maps (heightmap, textures, etc) and the generated terrain; higher values indicate lower accuracy but lower rendering overhead.
Base Map Distance	The maximum distance at which terrain textures will be displayed at full resolution. Beyond this distance, a lower resolution composite image will be used for efficiency.
Cast Shadows	Does the terrain cast shadows?
Material	The material used to render the terrain. This will affect how the color channels of a terrain texture are interpreted. See Enabling Textures for details. Available options are:
Built In Standard	This is the PBR (Physically-Based Rendering) material introduced in Unity 5.0. For each splat layer, you can use one texture for albedo and smoothness, one texture for normal and one scalar value to tweak the metalness. For more information on PBR and the Standard shader, see Standard Shader. Note: On Direct3D 9 in Shader Model 3.0, normal maps are not available if you have enabled directional lightmaps, Baked Global Illumination, real-time shadows, and shadowmasks. This is due to the limited number of samplers in Shaders.
Built In Legacy Diffuse	This is the legacy built-in terrain material from Unity 4.x and before. It uses Lambert (diffuse term only) lighting model and has optional normal map support.
Built In Legacy Specular	This built-in material uses BlinnPhong (diffuse and specular term) lighting model and has optional normal map support. You can specify the overall specular color and shininess for the terrain.
Custom	Use a custom material of your choice to render the terrain. This material should use a **shader** that is specialised for terrain rendering (e.g. it should handle texture splatting properly). We suggest you get a look at the source code of our built-in terrain shaders and make modifications on top of them.
Reflection Probes	How reflection probes are used on terrain. Only effective when using built-in standard material or a custom material which supports rendering with reflection. Available options are:
Off	Reflection probes are disabled, **skybox** will be used for reflection.
Blend Probes	Reflection probes are enabled. Blending occurs only between probes. Default reflection will be used if there are no reflection probes nearby, but no blending between default reflection and probe will occur.
Blend Probes And Skybox	Reflection probes are enabled. Blending occurs between probes or probes and default reflection.
Simple	Reflection probes are enabled, but no blending will occur between probes when there are two overlapping volumes.
Thickness	How much the terrain **collision** volume should extend along the negative Y-axis. Objects are considered colliding with the terrain from the surface to a depth equal to the thickness. This helps prevent high-speed moving objects from penetrating into the terrain without using expensive continuous **collision detection**.

Tree and Detail Objects

Property:	*Function:*
Draw	Should trees, grass and details be drawn?
Bake Light Probes__ For Trees**| If this option is enabled, Unity will create a Light Probe in the top of each tree and apply them to tree renderers for lighting. Otherwise trees are still affected by the ambient probe and** Light Probe groups__A component that enables you to add Light Probes to GameObjects in your scene. More info See in Glossary .	
Detail Distance	The distance (from camera) beyond which details will be culled.

Property:	*Function:*
Detail Density	The number of detail/grass objects in a given unit of area. The value can be set lower to reduce rendering overhead.
Tree Distance	The distance (from camera) beyond which trees will be culled.
Billboard Start	The distance (from camera) at which 3D tree objects will be replaced by **billboard** images.
Fade length	Distance over which trees will transition between **3D objects** and billboards.
Max Mesh__ Trees__	The maximum number of visible trees that will be represented as solid 3D meshes. Beyond this limit, trees will be replaced with billboards.

Wind Settings

Property:	*Function:*
Speed	The speed of the wind as it blows grass.
Size	The size of the "ripples" on grassy areas as the wind blows over them.
Bending	The degree to which grass objects are bent over by the wind.
Grass Tint	Overall color tint applied to grass objects.

Resolution

Property:	*Function:*
Terrain Width	Size of the terrain object in its X axis (in world units).
Terrain Length	Size of the terrain object in its Z axis (in world units).
Terrain Height	Difference in Y coordinate between the lowest possible heightmap value and the highest (in world units).
Heightmap Resolution	**Pixel** resolution of the terrain's heightmap (should be a power of two plus one, eg, 513 = 512 + 1).
Detail Resolution	Resolution of the map that determines the separate patches of details/grass. Higher resolution gives smaller and more detailed patches.
Detail Resolution Per Patch	Length/width of the square of patches renderered with a single draw call.
Control Texture Resolution	Resolution of the "splatmap" that controls the blending of the different terrain textures.
Base Texture Resolution	Resolution of the composite texture used on the terrain when viewed from a distance greater than the *Basemap Distance* (see above).

Heightmap Import/Export Buttons

The *Import Raw* and *Export Raw* buttons allow you to set or save the terrain's heightmap to an image file in the RAW grayscale format. RAW format can be generated by third party terrain editing tools (such as Bryce) and can also be opened, edited and saved by Photoshop. This allows for sophisticated generation and editing of terrains outside Unity.

2017–09–06 Page amended with limited editorial review

Cast Shadows flag added in 2017.2

Light Modes added in 5.6

Tree lightmap making updated in 5.6

Tree Editor

Leave feedback

SWITCH TO SCRIPTING

Unity provides a tool called **Tree Editor** that lets you design trees directly within the editor. This is very useful when you want to create detailed forests and jungles with different tree types and variations.

This section of the manual explains how to use the Tree Editor. Use the navigation column on the left-hand side of the page to view topics in this section.

SpeedTree

You can use SpeedTree Modeler from IDV, Inc. to create trees with advanced visual effects such as smooth **LOD** transition, fast **billboarding** and natural wind animation. See Unity documentation on SpeedTree for more information.

Building Your First Tree

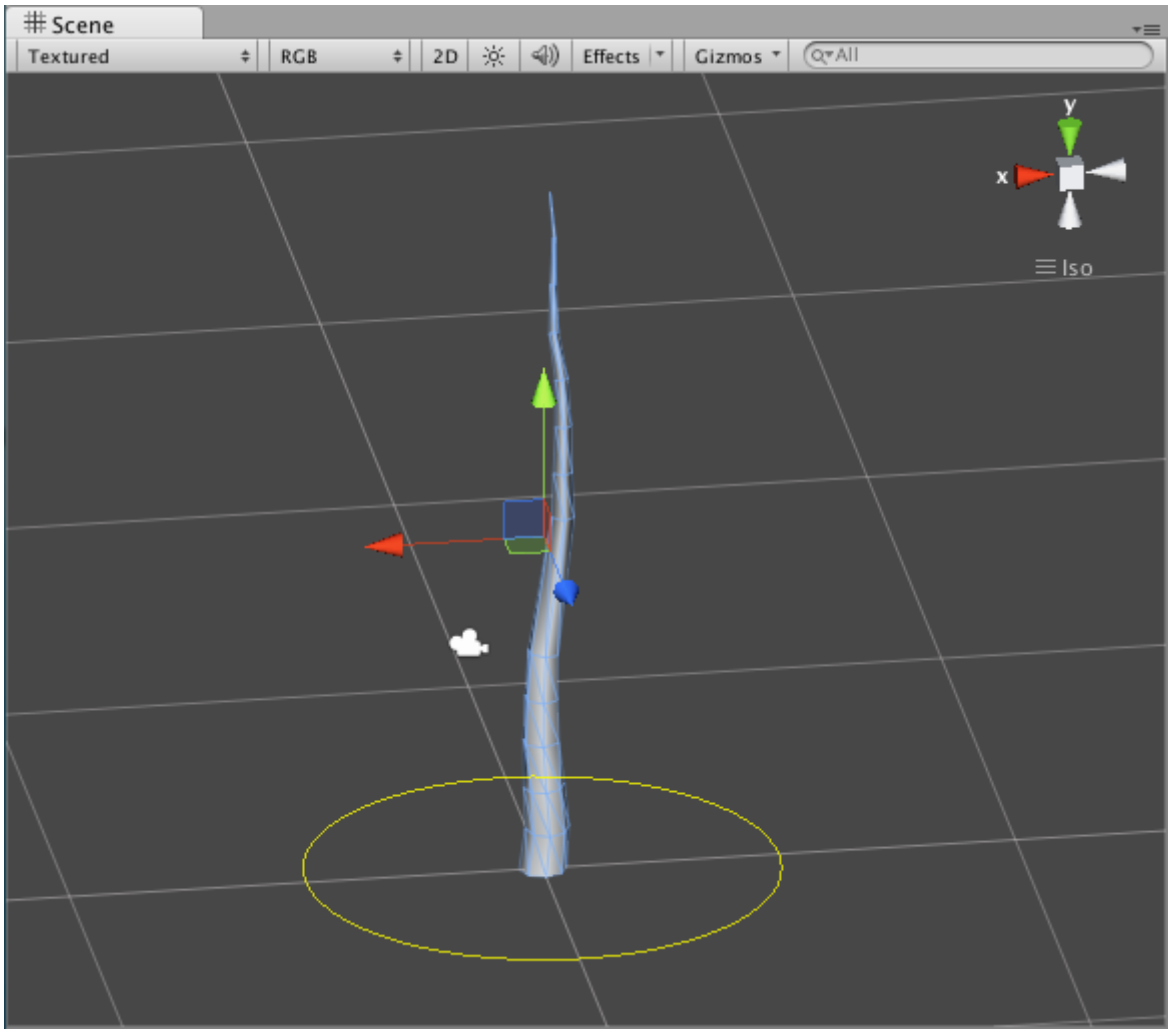
Leave feedback

We'll now walk you through the creation of your first tree with the Tree creation tool.

Adding a new Tree

To create a new **Tree** asset, select **GameObject > 3D Object > Tree**. You'll see a new Tree asset is created in your Project View, and instantiated in the currently open Scene. This new tree is very basic with only a single branch, so let's add some character to it.

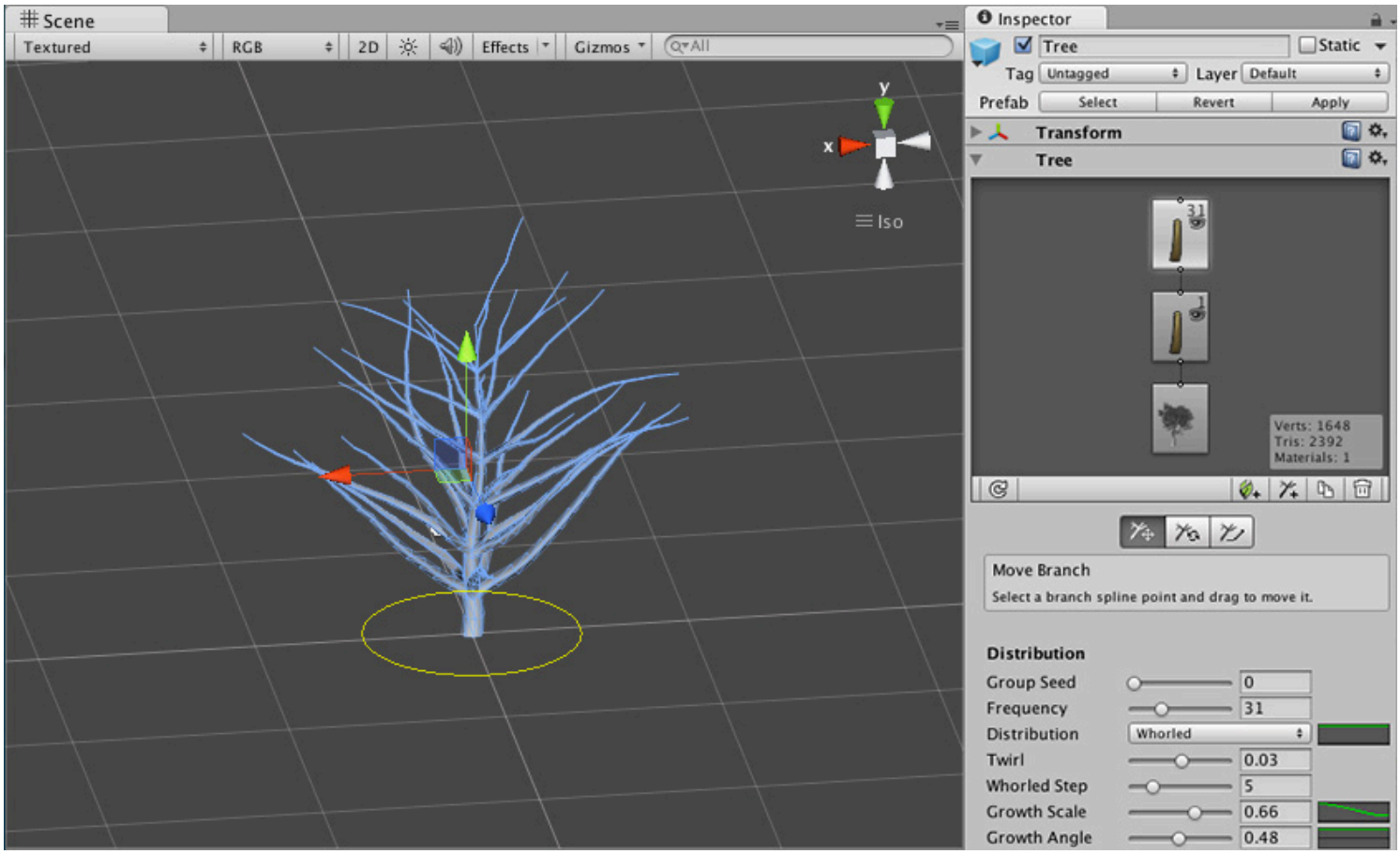
Adding Branches



A brand new tree in your scene

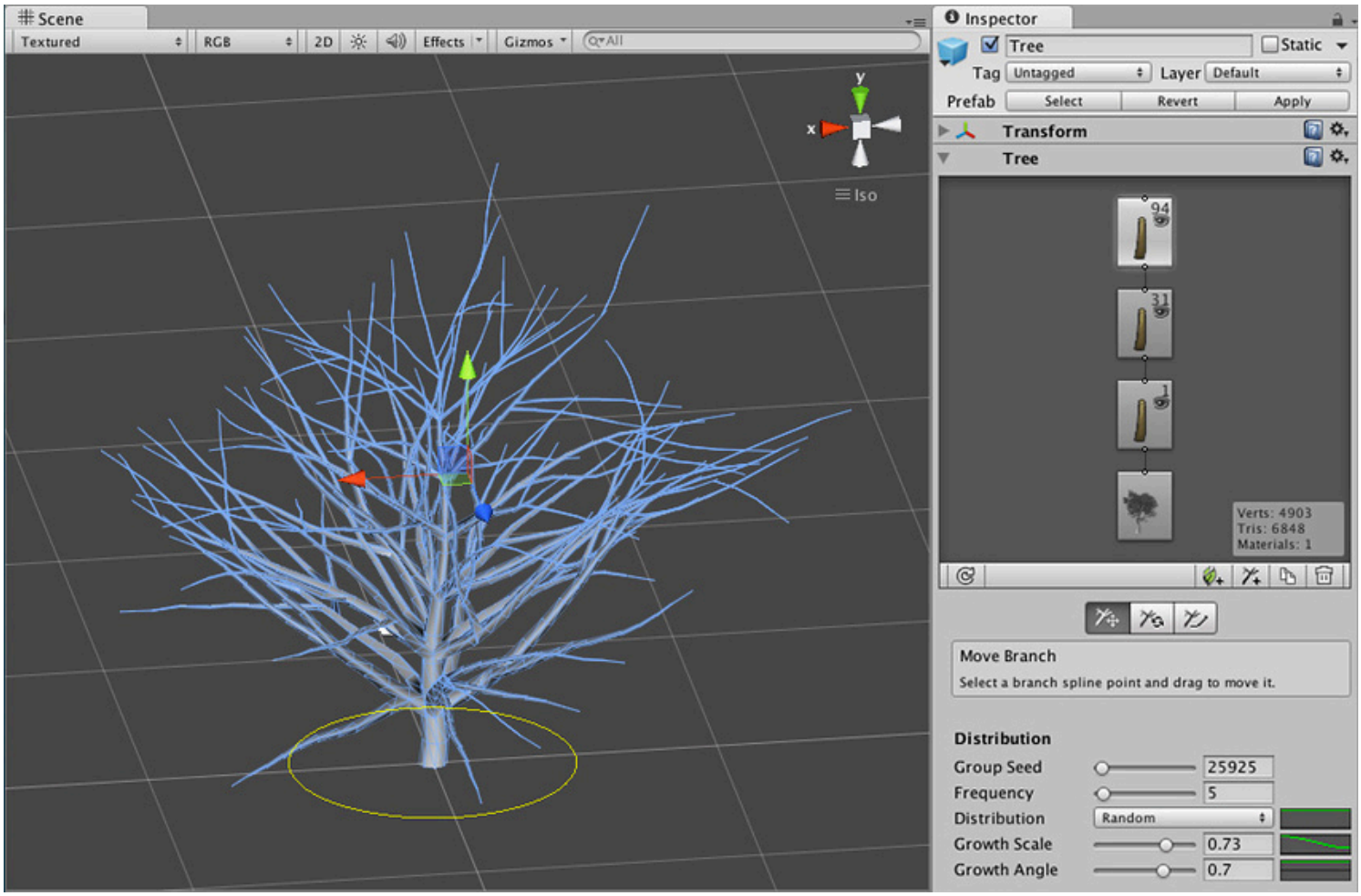
Select the tree to view the **Tree** window in the **Inspector**. This interface provides all the tools for shaping and sculpting your trees. You will see the **Tree Hierarchy** window with two nodes present: the **Tree Root** node and a single **Branch Group** node, which we'll call the trunk of the tree.

In the **Tree Hierarchy**, select the **Branch Group**, which acts as the trunk of the tree. Click on the **Add Branch Group** button and you'll see a new **Branch Group** appear connected to the Main Branch. Now you can play with the settings in the **Branch Group Properties** to see alterations of the branches attached to the tree trunk.



Adding branches to the tree trunk.

After creating the branches that are attached to the trunk, we can now add smaller twigs to the newly created branches by attaching another **Branch Group** node. Select the secondary **Branch Group** and click the **Add Branch Group** button again. Tweak the values of this group to create more branches that are attached to the secondary branches.

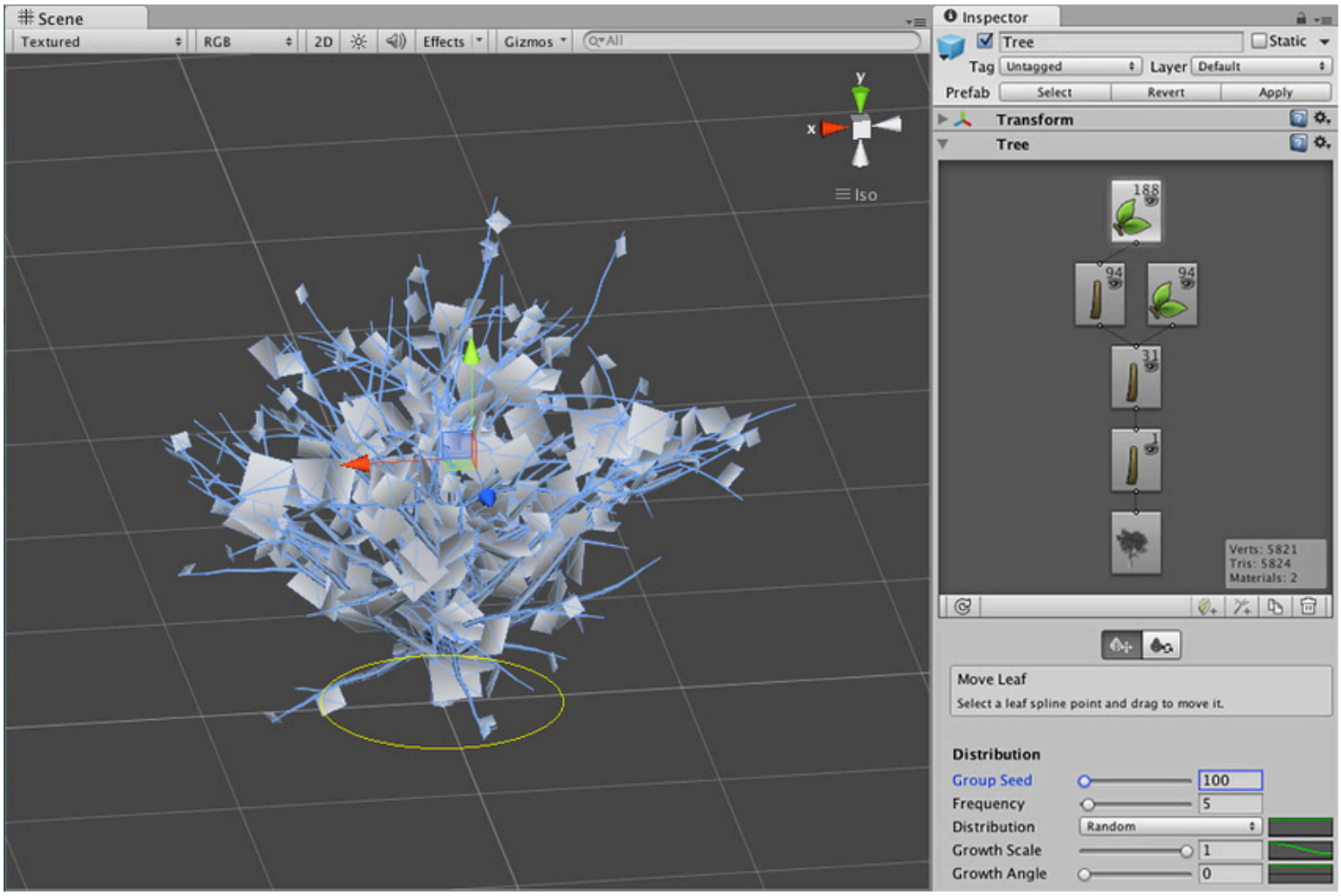


Adding branches to the secondary branches.

Now the tree's branch structure is in place. Our game doesn't take place in the winter time, so we should also add some **Leaves** to the different branches, right?

Adding Leaves

We decorate our tree with leaves by adding **Leaf Groups**, which basically work the same as the Branch groups we've already used. Select your secondary Branch Group node and then click the **Add Leaf Group** button. If you're really hardcore, you can add another leaf group to the tiniest branches on the tree as well.



Leaves added to the secondary and smallest branches

Right now the leaves are rendered as opaque planes. This is because we want to adjust the leaves' values (size, position, rotation, etc.) before we add a material to them. Tweak the Leaf values until you find some settings you like.

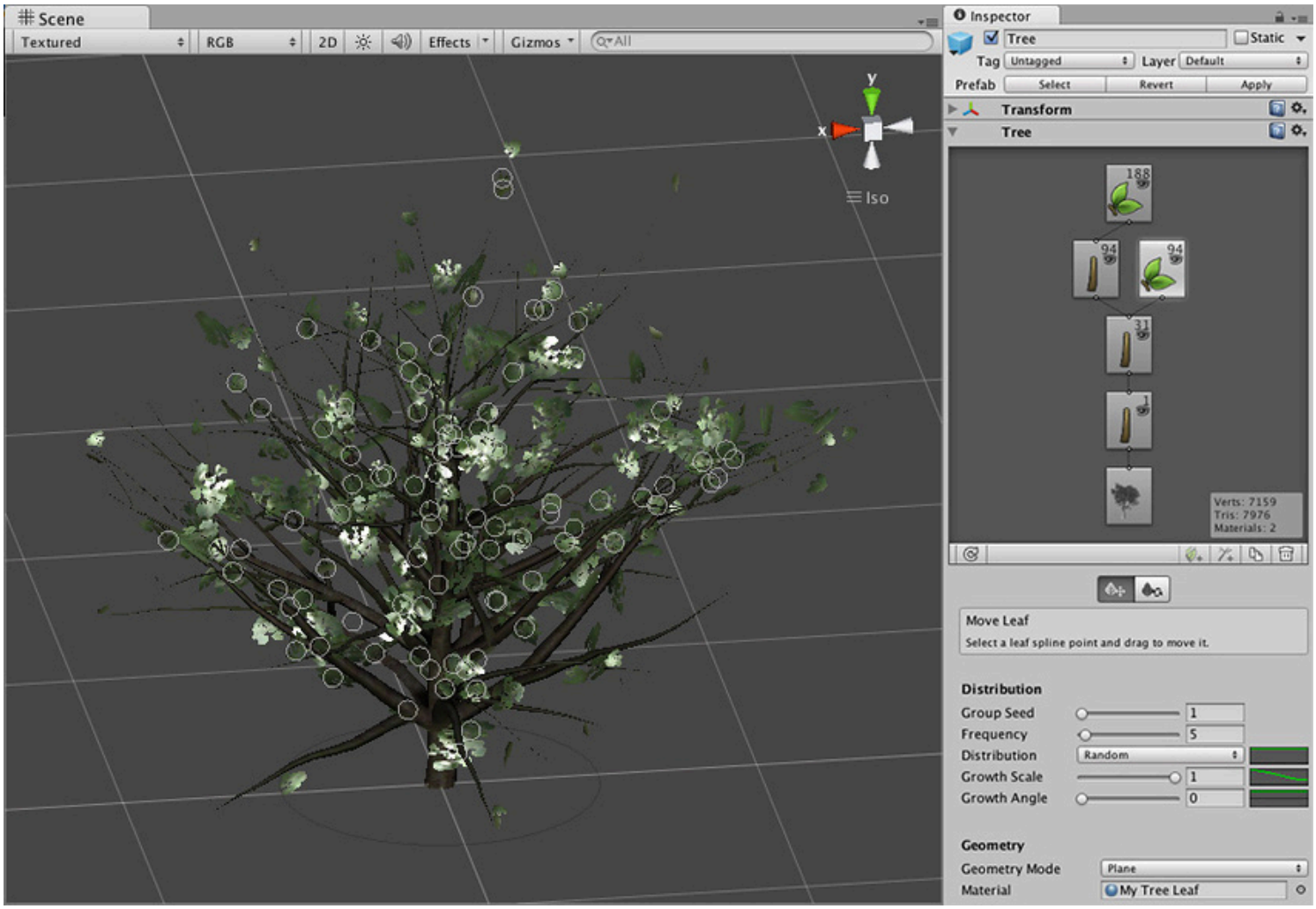
Adding Materials

In order to make our tree realistic looking, we need to apply **Materials** for the branches and the leaves. Create a new Material in your project using **Assets > Create > Material**. Rename it to "My Tree Bark", and choose **Nature > Tree Creator Bark** from the Shader drop-down. From here you can assign the **Textures** provided in the Tree Creator Package to the Base, Normalmap, and Gloss properties of the Bark Material. We recommend using the texture "BigTree_bark_diffuse" for the Base and Gloss properties, and "BigTree_bark_normal" for the Normalmap property.

Now we'll follow the same steps for creating a Leaf Material. Create a new Material and assign the shader as **Nature > Tree Creator Leaves**. Assign the texture slots with the leaf textures from the Tree Creator Package.

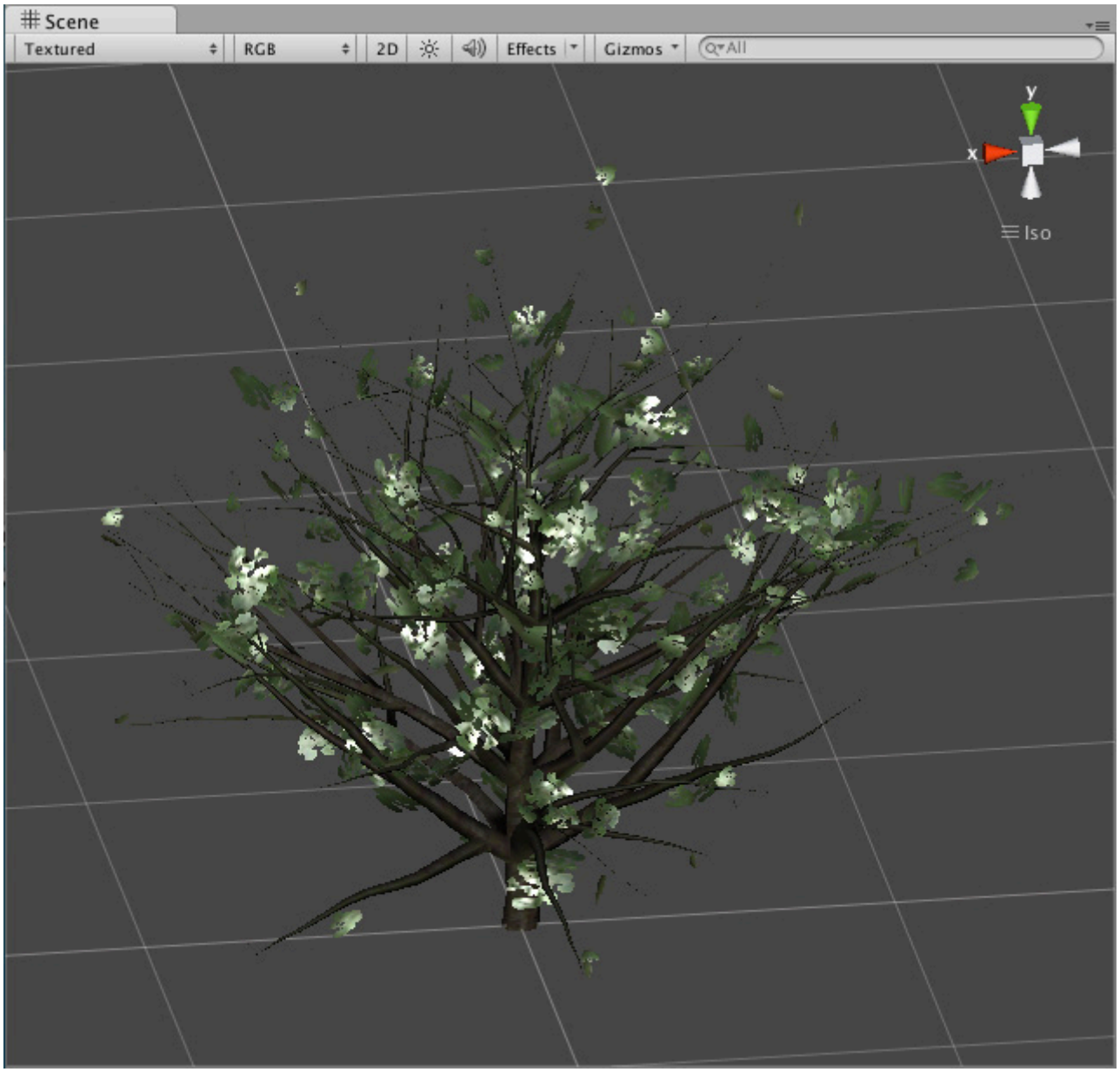
Material for the Leaves

When both Materials are created, we'll assign them to the different Group Nodes of the Tree. Select your Tree and click any Branch or Leaf node, then expand the **Geometry** section of the **Branch Group Properties**. You will see a Material assignment slot for the type of node you've selected. Assign the relevant Material you created and view the results.



Setting the leaves material

To finish off the tree, assign your Materials to all the **Branch** and **Leaf Group** nodes in the Tree. Now you're ready to put your first tree into a game!



Tree with materials on leaves and branches.

Hints.

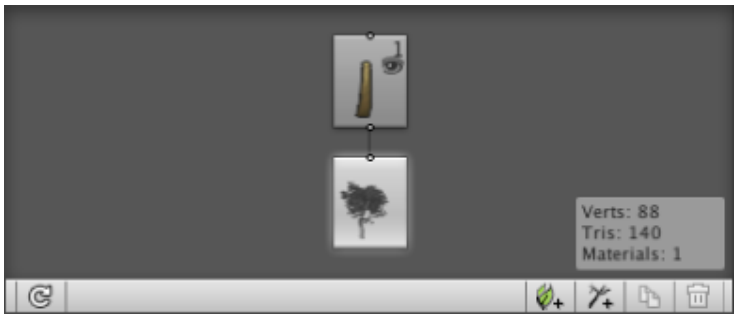
Creating trees is a trial and error process.
Don't create too many leaves/branches as this can affect the performance of your game.
Check the alpha maps guide for creating custom leaves.

Tree Basics

Leave feedback

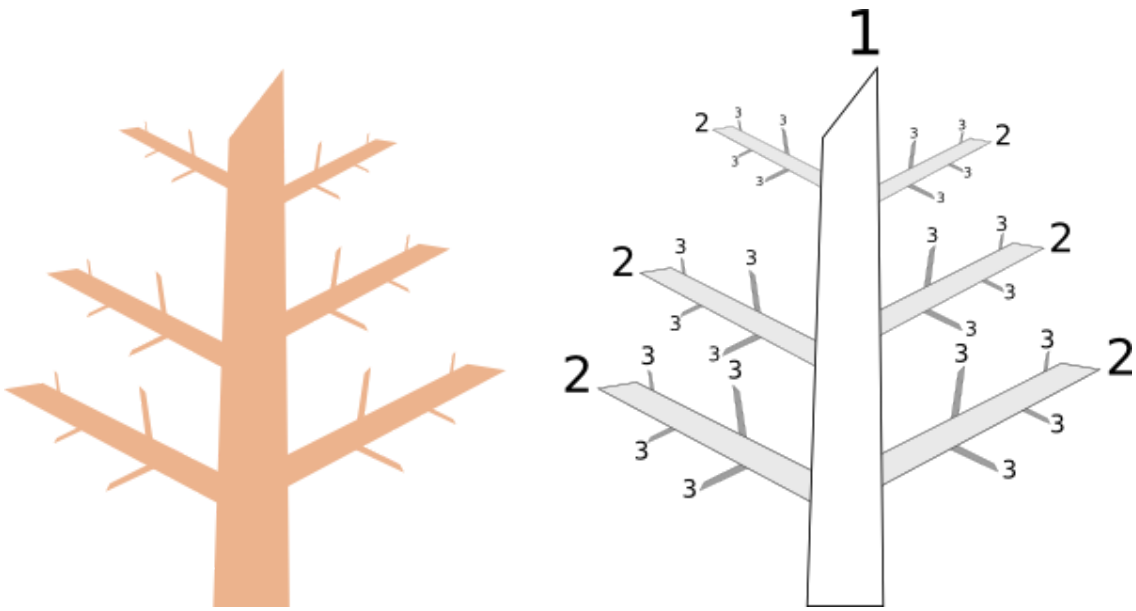
With the Tree Creator package imported, you can select **GameObject > 3D Object > Tree** to add a new tree to the scene (this will also create a new Tree asset in the Project view). The tree this produces is initially little more than a single stalk with no leaves or branches. However, you will notice in the inspector that the object has a Tree component attached, and this will allow you to design the tree to your liking.

At the top of the Tree **component inspector** is the *tree structure editor* where the basic arrangement of branches and leaves is specified.

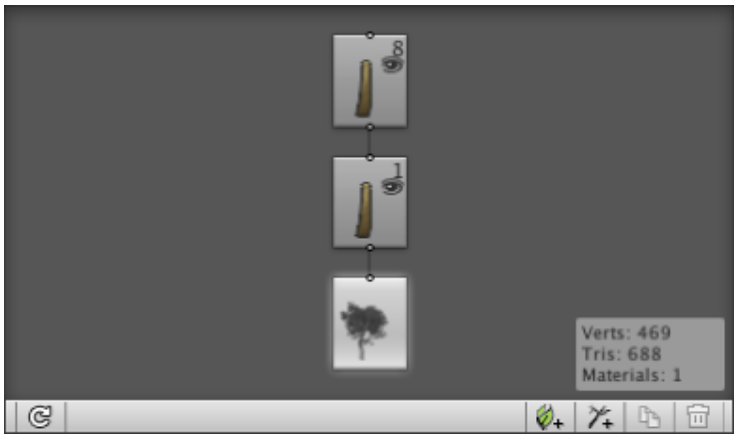


Tree Structure editor

It is important to understand the concept of *tree levels* when working with the editor. The trunk has branches which, in turn, have sub-branches; this branching process continues until the terminal twig are produced. The trunk is regarded as the first level of the tree, and then any branches growing directly from the trunk comprise the second level. Any branches that grow from second level branches together form the third level, and so on.



This notion of levels is reflected in the tree editor. Consider the following tree structure, for example:-

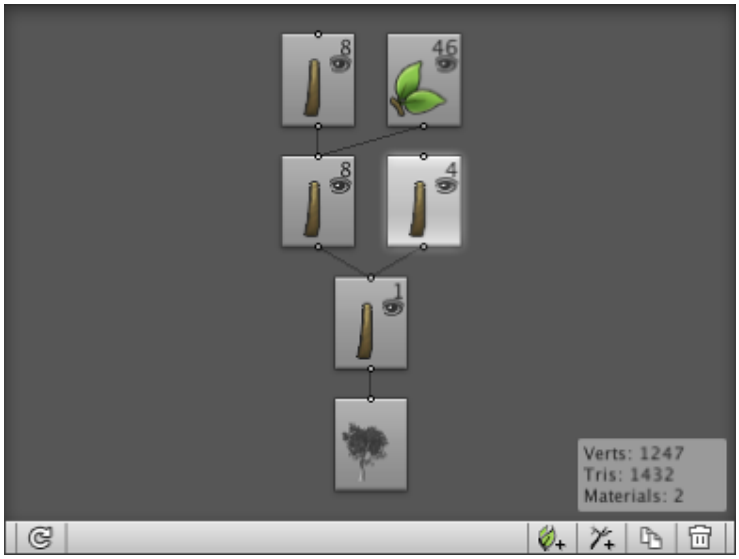


The icons are connected by lines to show the branching levels of the tree. The icon right at the bottom (with the tree picture) denotes the “root” of the tree. When this icon is selected, the properties in the inspector panel below are the ones that apply to the tree as a whole. From this root extend the first and second levels of branching. The icons show several pieces of information:-

The main picture shows which kind of element it is. The number in the top-right corner is the number of branches that exist at that level of the tree, as set by the *Frequency* property in the inspector. With a given icon selected, changing the *Frequency* value will change the number of branches at that level. The eye image just below the number denotes the visibility of the branches in the **scene view**; you click the eye to toggle visibility on or off.

The arrangement of **branch groups** can be edited using the controls at the bottom-right of the tree editor:-

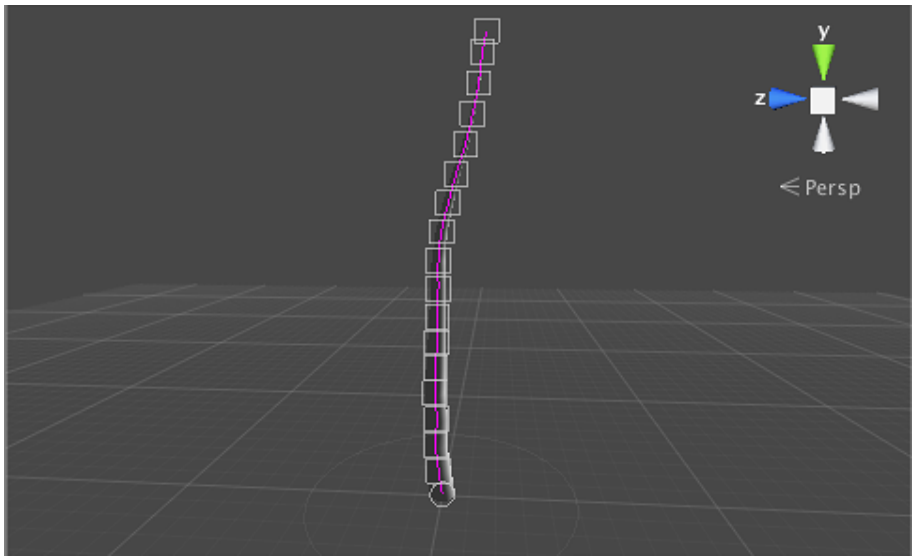
Going from left to right, the first tool adds *leaf* groups to the tree. Leaves are arranged in levels as are branches but unlike branches, leaves cannot further subdivide into more levels. The second tool adds a new branch group at the current level (ie, it creates a new “child” for the selected branch icon). The third tool duplicates whichever group is selected while the fourth deletes a group from the tree. It is possible to have several groups at each level of a tree as in the following example:-



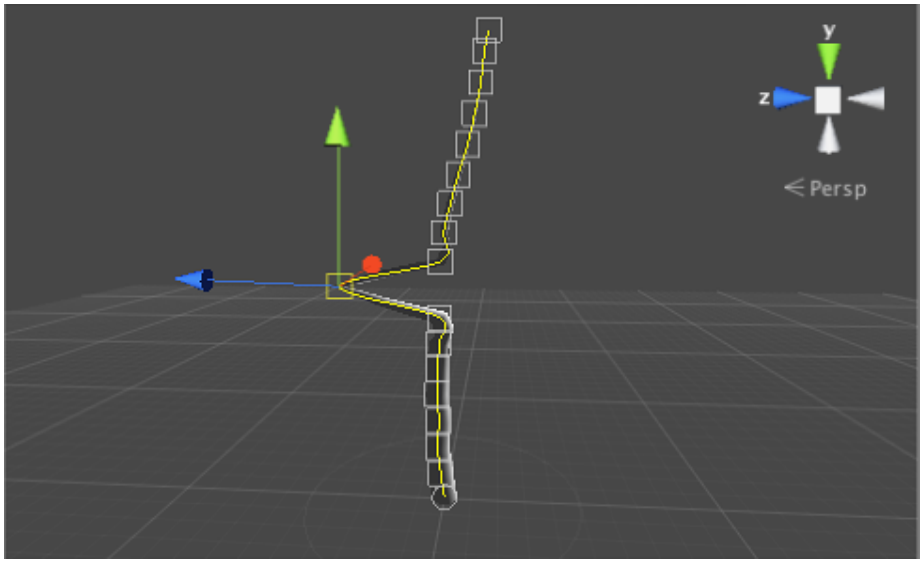
This tree has a main trunk from which two different branch groups grow. The first has its own sub-groups of branches and leaves, while the second just has bare branches. The separate groups at a given level can each have their properties set differently in the inspector so you could, say, have a large number of short twigs sprouting from the trunk along with a smaller number of main branches.

Hand Editing Branches and Leaves

When a branch is selected in the tree structure view, it will also be highlighted in the **scene** view, as with this "tree" (which is just a bare trunk for now).



The tree's single branch is shown with a number of boxes overlaid on the view. The boxes represent *control points* along the length of the branch (ie, the center line of the branch passes through all the points but is also smoothly curved between them). You can click and drag any of the boxes to move the control points and thus change the shape of the branch.



Moving control points is actually just the first of three options available on the hand editing **toolbar**.

The second tool allows you to bend the branch by rotating it at a given control point. The third tool allows you to start with the mouse at a given control point and from there on draw the branch freehand. The branching is still controlled

from the structure view - only the shapes of branches can be redrawn. If a leaf group is selected in the structure view then a corresponding toolbar gives you options to move or rotate leaves around their parent branch.

Note that some properties in the tree creator's inspector are related to *procedural* generation of trees (ie, the computer generates the shape itself randomly) and these will be disabled after you have hand edited the tree. There is a button which will restore a tree to procedural status but this will undo any edits you have made by hand.

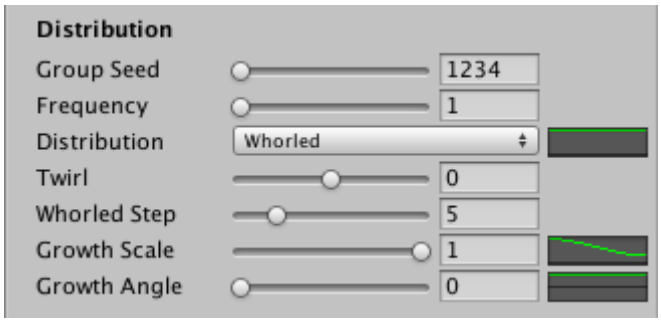
Branch Group Properties

Leave feedback

Branch groups node is responsible for generating branches and fronds. Its properties appear when you have selected a branch, frond or branch + frond node.

Distribution

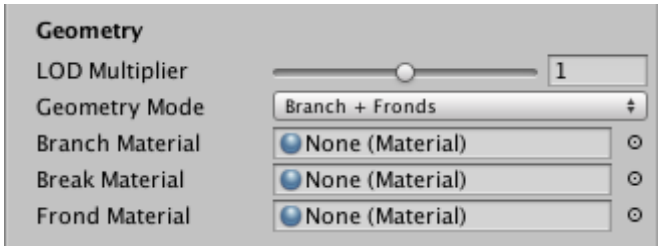
Adjusts the count and placement of branches in the group. Use the curves to fine tune position, rotation and scale. The curves are relative to the parent branch or to the area spread in case of a trunk.



Group Seed	The seed for this group of branches. Modify to vary procedural generation.
Frequency	Adjusts the number of branches created for each parent branch.
Distribution	The way the branches are distributed along their parent.
Twirl	Twirl around the parent branch.
Whorled Step	Defines how many nodes are in each whorled step when using Whorled distribution. For real plants this is normally a Fibonacci number.
Growth Scale	Defines the scale of nodes along the parent node. Use the curve to adjust and the slider to fade the effect in and out.
Growth Angle	Defines the initial angle of growth relative to the parent. Use the curve to adjust and the slider to fade the effect in and out.

Geometry

Select what type of geometry is generated for this **branch group** and which materials are applied. **LOD Multiplier** allows you to adjust the quality of this group relative to tree's **LOD Quality**.

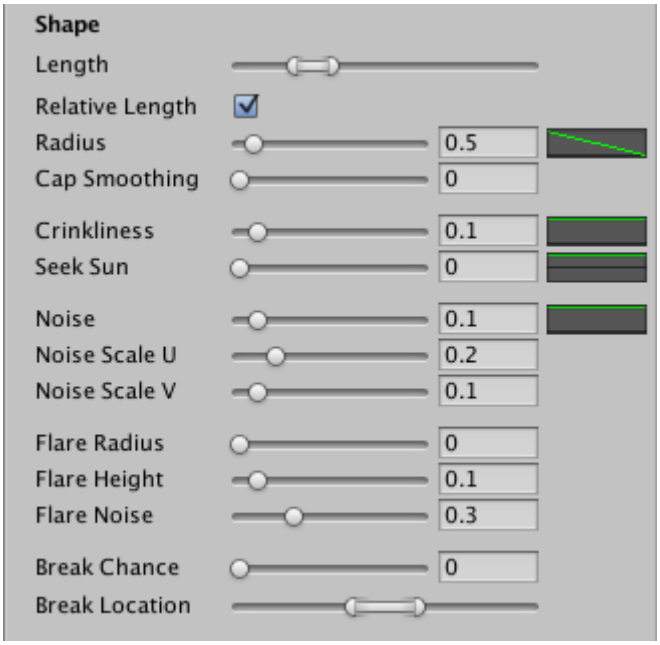


LOD Multiplier	Adjusts the quality of this group relative to tree's **LOD** Quality, so that it is of either higher or lower quality than the rest of the tree.
Geometry Mode	Type of geometry for this branch group: Branch Only, Branch + Fronds, Fronds Only.
Branch Material	The primary material for the branches.

Break Material	Material for capping broken branches.
Frond Material	Material for the fronds.

Shape

Adjusts the shape and growth of the branches. Use the curves to fine tune the shape, all curves are relative to the branch itself.



Length	Adjusts the length of the branches.
Relative Length	Determines whether the radius of a branch is affected by its length.
Radius	Adjusts the radius of the branches, use the curve to fine-tune the radius along the length of the branches.
Cap Smoothing	Defines the roundness of the cap/tip of the branches. Useful for cacti.
Growth	Adjusts the growth of the branches.
Crinkliness	Adjusts how crinkly/crooked the branches are, use the curve to fine-tune.
Seek Sun	Use the curve to adjust how the branches are bent upwards/downwards and the slider to change the scale.
Surface Noise	Adjusts the surface noise of the branches.
Noise	Overall noise factor, use the curve to fine-tune.
Noise Scale U	Scale of the noise around the branch, lower values will give a more wobbly look, while higher values gives a more stochastic look.
Noise Scale V	Scale of the noise along the branch, lower values will give a more wobbly look, while higher values gives a more stochastic look.
Flare	Defines a flare for the trunk.
Flare Radius	The radius of the flares, this is added to the main radius, so a zero value means no flares.

Flare Height	Defines how far up the trunk the flares start.
Flare Noise	Defines the noise of the flares, lower values will give a more wobbly look, while higher values gives a more stochastic look.
Breaking	Controls the breaking of branches.
Break Chance	Chance of a branch breaking, i.e. 0 = no branches are broken, 0.5 = half of the branches are broken, 1.0 = all the branches are broken.
Break Location	This range defines where the branches will be broken. Relative to the length of the branch.

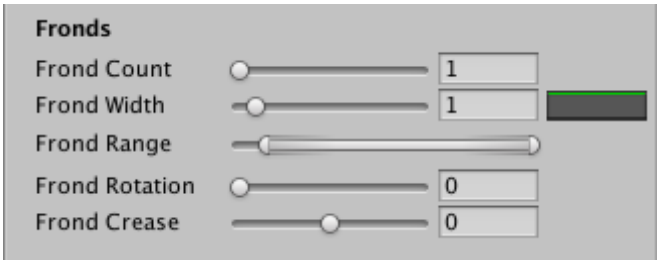
These properties are for child branches only, not trunks.

Welding ** | Defines the welding of branches onto their parent branch. Only valid for secondary branches.**

Weld Length	Defines how far up the branch the weld spread starts.
Spread Top	Weld's spread factor on the top-side of the branch, relative to it's parent branch. Zero means no spread.
Spread Bottom	Weld's spread factor on the bottom-side of the branch, relative to it's parent branch. Zero means no spread.

Fronds

Here you can adjust the number of fronds and their properties. This tab is only available if you have Frond geometry enabled in the **Geometry** tab.



Frond Count	Defines the number of fronds per branch. Fronds are always evenly spaced around the branch.
Frond Width	The width of the fronds, use the curve to adjust the specific shape along the length of the branch.
Frond Range	Defines the starting and ending point of the fronds.
Frond Rotation	Defines rotation around the parent branch.
Frond Crease	Adjust to crease / fold the fronds.

Wind

Adjusts the parameters used for animating this group of branches. The wind zones are only active in Play Mode.

Main Wind	Primary wind effect. This creates a soft swaying motion and is typically the only parameter needed for primary branches.
Edge Turbulence	Turbulence along the edge of fronds. Useful for ferns, palms, **etc**.
Create Wind Zone	Creates a Wind Zone.

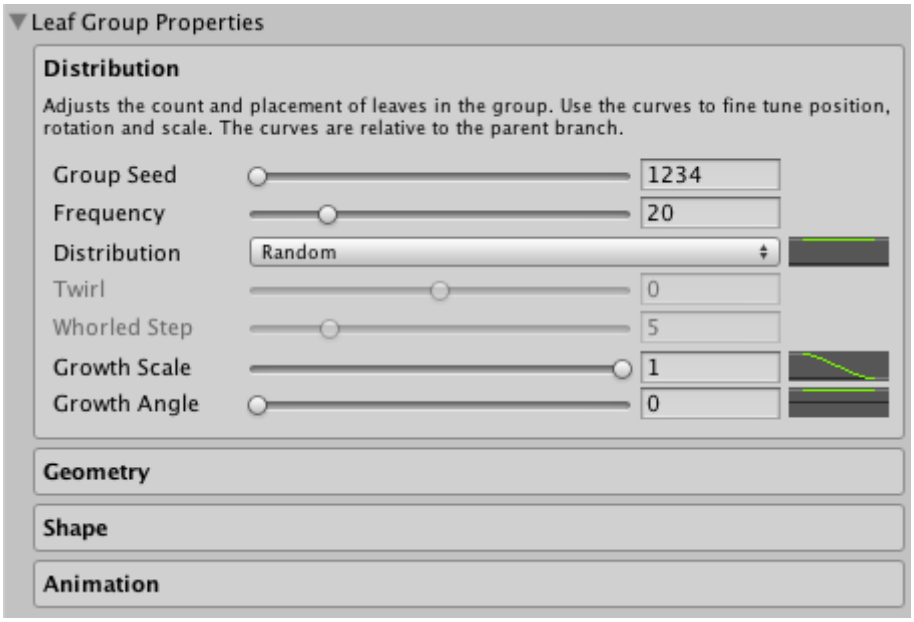
Leaf Group Properties

Leave feedback

Leaf groups generate leaf geometry. Either from primitives or from user created meshes.

Distribution

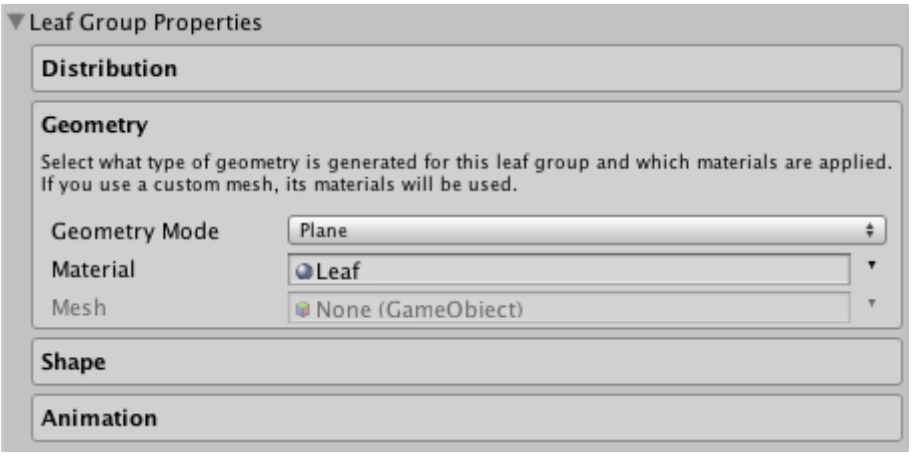
Adjusts the count and placement of leaves in the group. Use the curves to fine tune position, rotation and scale. The curves are relative to the parent branch.



Group Seed	The seed for this group of leaves. Modify to vary procedural generation.
Frequency	Adjusts the number of leaves created for each parent branch.
Distribution	Select the way the leaves are distributed along their parent.
Twirl	Twirl around the parent branch.
Whorled Step	Defines how many nodes are in each whorled step when using Whorled distribution. For real plants this is normally a Fibonacci number.
Growth Scale	Defines the scale of nodes along the parent node. Use the curve to adjust and the slider to fade the effect in and out.
Growth Angle	Defines the initial angle of growth relative to the parent. Use the curve to adjust and the slider to fade the effect in and out.

Geometry

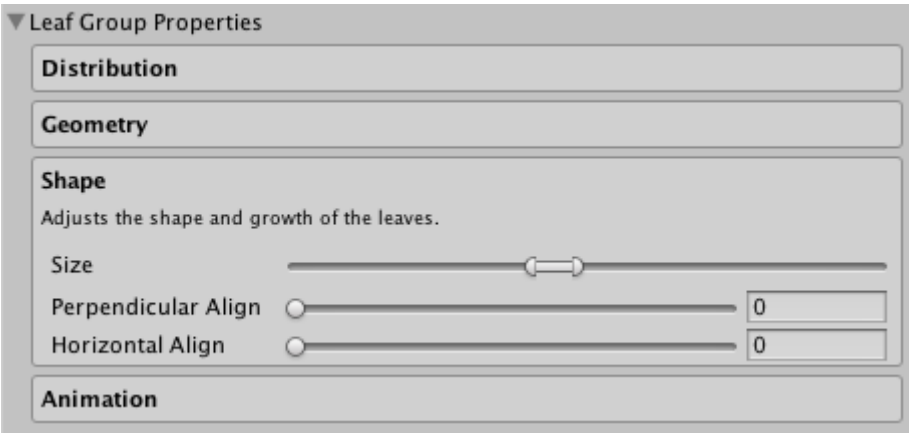
Select what type of geometry is generated for this leaf group and which materials are applied. If you use a custom mesh, its materials will be used.



Geometry Mode	The type of geometry created. You can use a custom mesh, by selecting the Mesh option, ideal for flowers, fruits, **etc**.
Material	Material used for the leaves.
Mesh	Mesh used for the leaves.

Shape

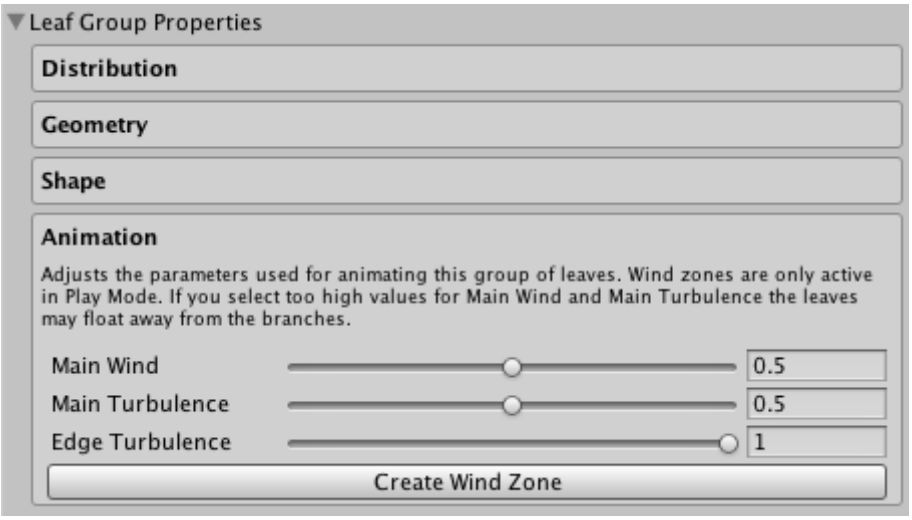
Adjusts the shape and growth of the leaves.



Size	Adjusts the size of the leaves, use the range to adjust the minimum and the maximum size.
Perpendicular Align	Adjusts whether the leaves are aligned perpendicular to the parent branch.
Horizontal Align	Adjusts whether the leaves are aligned horizontally.

Animation

Adjusts the parameters used for animating this group of leaves. **Wind zones** are only active in Play Mode. If you select too high values for Main Wind and Main Turbulence the leaves may float away from the branches.



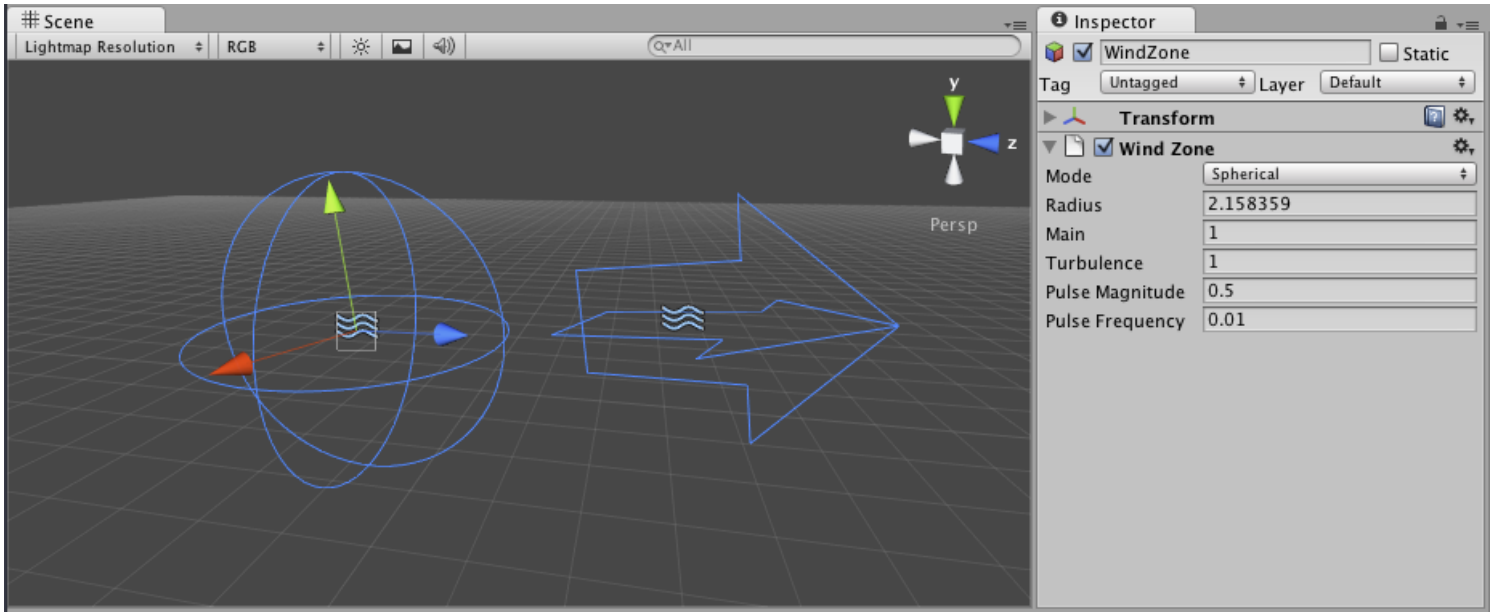
Main Wind	Primary wind effect. Usually this should be kept as a low value to avoid leaves floating away from the parent branch.
Main Turbulence	Secondary turbulence effect. For leaves this should usually be kept as a low value.
Edge Turbulence	Defines how much wind turbulence occurs along the edges of the leaves.

Tree - Wind Zones

Leave feedback

SWITCH TO SCRIPTING

Wind Zones add realism to the trees you create by making them wave their branches and leaves as if blown by the wind.



To the left a Spherical Wind Zone, to the right a Directional Wind zone.

Properties

Property:	*Function:*
Mode	
Spherical	Wind zone only has an effect inside the radius, and has a falloff from the center towards the edge.
Directional	Wind zone affects the entire **scene** in one direction.
Radius	Radius of the Spherical Wind Zone (only active if the mode is set to Spherical).
Main	The primary wind force. Produces a softly changing wind pressure.
Turbulence	The turbulence wind force. Produces a rapidly changing wind pressure.
Pulse Magnitude	Defines how much the wind changes over time.
Pulse Frequency	Defines the frequency of the wind changes.

Details

Wind Zones are used only by the tree creator for animating leaves and branches. This can help your scenes appear more natural and allows forces (such as explosions) within the game to look like they are interacting with the trees. For more information about how a tree works, just visit the tree class page.

Using Wind Zones in Unity

Using **Wind Zones** in Unity is really simple. First of all, to create a new **wind zone** just click on **Game Object > 3D Object > Wind Zone**.

Place the wind zone (depending on the type) near the trees created with the tree creator and watch it interact with your trees!.

Note: If the wind zone is Spherical you should place it so that the trees you want to blow are within the sphere's radius. If the wind zone is directional it doesn't matter where in the scene you place it.

Hints

To produce a softly changing general wind:
Create a directional wind zone.
Set Wind Main to 1.0 or less, depending on how powerful the wind should be.
Set Turbulence to 0.1.
Set Pulse Magnitude to 1.0 or more.
Set Pulse Frequency to 0.25.
To create the effect of a helicopter passing by:
Create a spherical wind zone.
Set Radius to something that fits the size of your helicopter
Set Wind Main to 3.0
Set Turbulence to 5.0
Set Pulse Magnitude to 0.1
Set Pulse Frequency to 1.0
Attach the wind zone to a **GameObject** resembling your helicopter.
To create the effect of an explosion:
Do the same as with the helicopter, but fade the Wind Main and Turbulence quickly to make the effect wear off.

2017–09–19 Page amended with limited editorial review

GameObject menu changed in Unity 4.6

Particle Systems

Leave feedback

In a 3D game, most characters, props and scenery elements are represented as **meshes**, while a 2D game uses **sprites** for these purposes. Meshes and sprites are the ideal way to depict “solid” objects with a well-defined shape. There are other entities in games, however, that are fluid and intangible in nature and consequently difficult to portray using meshes or sprites. For effects like moving liquids, smoke, clouds, flames and magic spells, a different approach to graphics known as **particle systems** can be used to capture the inherent fluidity and energy. This section explains Unity’s particle systems and what they can be used for.

What is a Particle System?

Leave feedback

Particles are small, simple images or meshes that are displayed and moved in great numbers by a **particle system**. Each particle represents a small portion of a fluid or amorphous entity and the effect of all the particles together creates the impression of the complete entity. Using a smoke cloud as an example, each particle would have a small smoke texture resembling a tiny cloud in its own right. When many of these mini-clouds are arranged together in an area of the **scene**, the overall effect is of a larger, volume-filling cloud.

Dynamics of the System

Each particle has a predetermined *lifetime*, typically of a few seconds, during which it can undergo various changes. It begins its life when it is generated or *emitted* by its particle system. The system emits particles at random positions within a region of space shaped like a sphere, hemisphere, cone, box or any arbitrary **mesh**. The particle is displayed until its time is up, at which point it is removed from the system. The system's *emission rate* indicates roughly how many particles are emitted per second, although the exact times of emission are randomized slightly. The choice of emission rate and average particle lifetime determine the number of particles in the "stable" state (ie, where emission and particle death are happening at the same rate) and how long the system takes to reach that state.

Dynamics of Particles

The emission and lifetime settings affect the overall behaviour of the system but the individual particles can also change over time. Each one has a **velocity** vector that determines the direction and distance the particle moves with each frame update. The velocity can be changed by **forces** and **gravity** applied by the system itself or when the particles are blown around by a **wind zone** on a **Terrain**. The color, size and rotation of each particle can also change over its lifetime or in proportion to its current speed of movement. The color includes an alpha (transparency) **component**, so a particle can be made to fade gradually in and out of existence rather than simply appearing and disappearing abruptly.

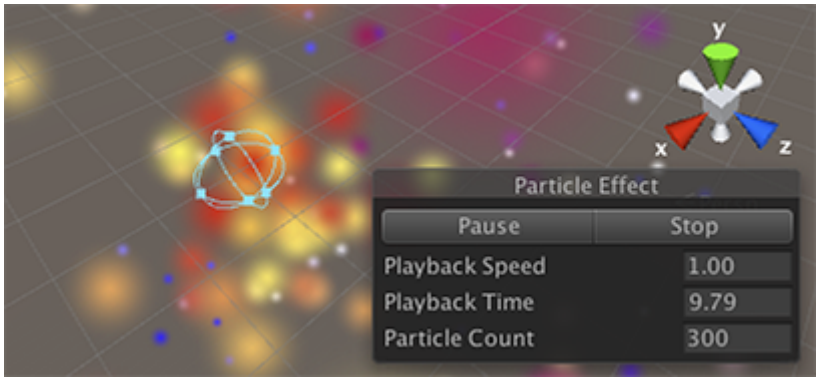
Used in combination, particle dynamics can be used to simulate many kinds of fluid effects quite convincingly. For example, a waterfall can be simulated by using a thin emission shape and letting the water particles simply fall under gravity, accelerating as they go. Smoke from a fire tends to rise, expand and eventually dissipate, so the system should use an upward force on the smoke particles and increase their size and transparency over their lifetimes.

Using Particle Systems in Unity

Leave feedback

Unity implements Particle Systems with a component, so placing a Particle System in a Scene is a matter of adding a pre-made GameObject (menu: **GameObject** > **Effects** > **Particle System**) or adding the component to an existing GameObject (menu: **Component** > **Effects** > **Particle System**). Because the component is quite complicated, the Inspector is divided into a number of collapsible sub-sections or **modules** that each contain a group of related properties. Additionally, you can edit one or more systems at the same time using a separate Editor window accessed via the **Open Window** button in the Inspector. See documentation on the Particle System component and individual Particle System modules to learn more.

When a GameObject with a Particle System is selected, the **Scene view** contains a small **Particle Effect** panel, with some simple controls that are useful for visualising changes you make to the system's settings.



The **Playback Speed** allows you to speed up or slow down the particle simulation, so you can quickly see how it looks at an advanced state. The **Playback Time** indicates the time elapsed since the system was started; this may be faster or slower than real time depending on the playback speed. The **Particle Count** indicates how many particles are currently in the system. The playback time can be moved backwards and forwards by clicking on the **Playback Time** label and dragging the mouse left and right. The buttons at the top of the panel can be used to pause and resume the simulation, or to stop it and reset to the initial state.

Varying properties over time

Many of the numeric properties of particles or even the whole Particle System can vary over time. Unity provides several different methods of specifying how this variation happens:

Constant: The property's value is fixed throughout its lifetime.
Curve: The value is specified by a curve/graph.
Random Between Two Constants: Two constant values define the upper and lower bounds for the value; the actual value varies randomly over time between those bounds.
Random Between Two Curves: Two curves define the upper and lower bounds of the the value at a given point in its lifetime; the current value varies randomly between those bounds.
Similarly, the **Start Color** property in the main module has the following options:

Color: The particle start color is fixed throughout the system's lifetime.
Gradient: Particles are emitted with a start color specified by a gradient, with the gradient representing the lifetime of the Particle System.
Random Between Two Colors: The starting particle color is chosen as a random linear **interpolation** between the two given colors.

Random Between Two Gradients: Two colors are picked from the given Gradients at the point corresponding to the current age of the system; the starting particle color is chosen as a random linear interpolation between these colors.

For other color properties, such as **Color over Lifetime**, there are two separate options:

Gradient: The color value is taken from a gradient which represents the lifetime of the Particle System.

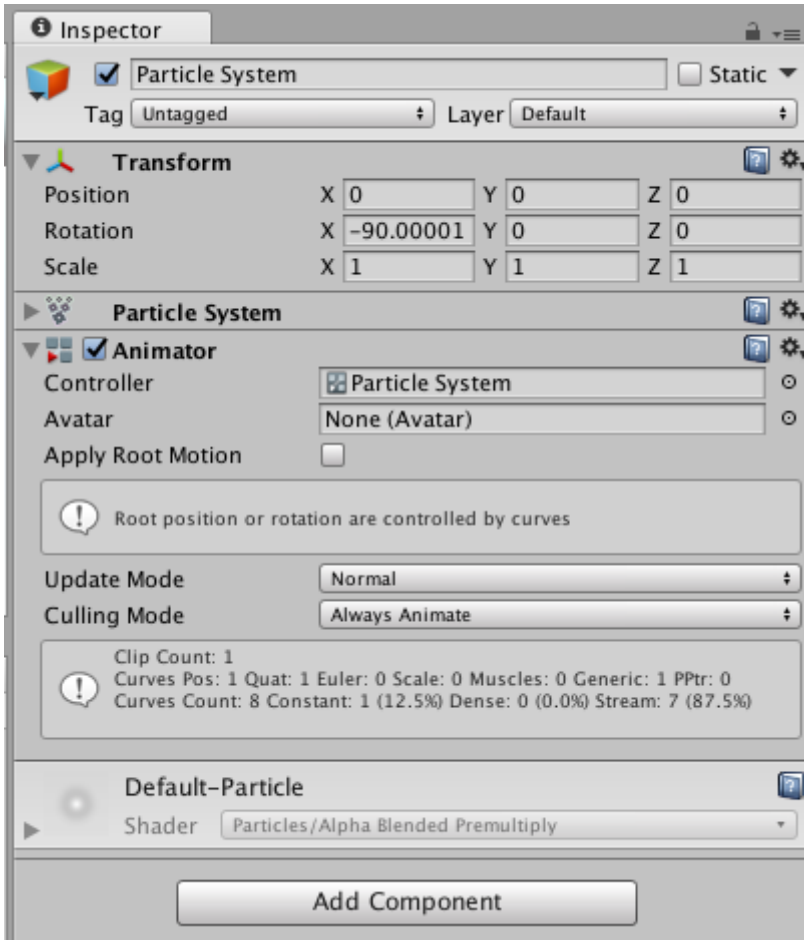
Random Between Two Gradients: Two colors are picked from the given gradients at the point corresponding to the current age of the Particle System; the color value is chosen as a random linear interpolation between these colors.

Color properties in various modules are multiplied together per channel to calculate the final particle color result.

Animation bindings

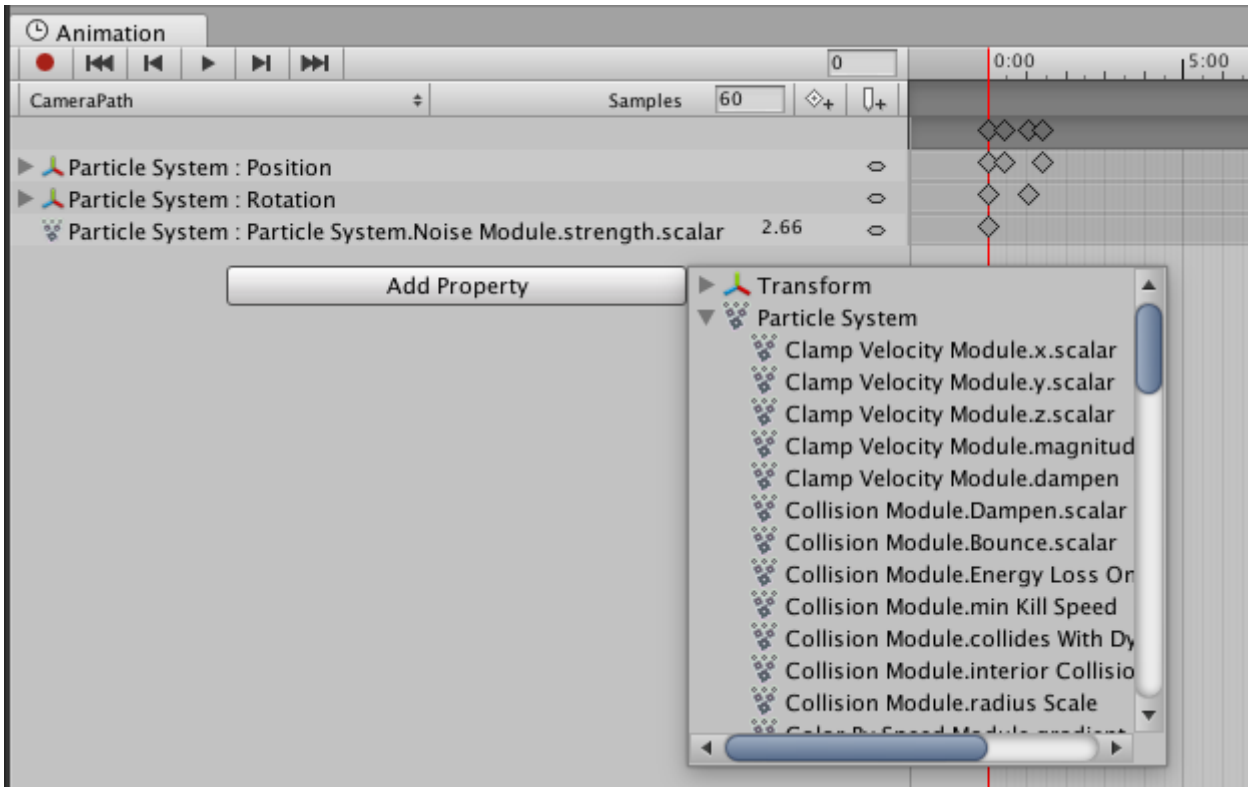
All particle properties are accessible by the Animation system, meaning you can **keyframe** them in and control them from your animations.

To access the Particle System's properties, there must be an **Animator component** attached to the Particle System's GameObject. An Animation Controller and an Animation are also required.



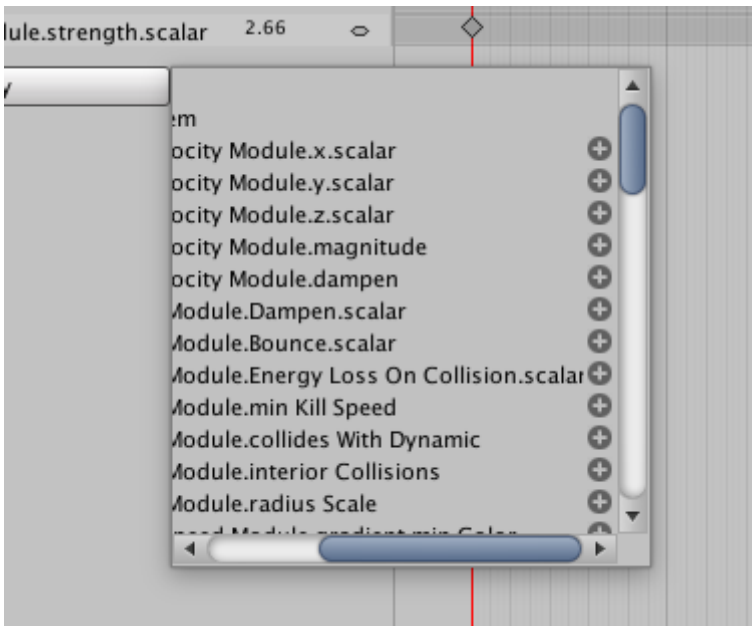
To animate a Particle System, add an Animator Component, and assign an Animation Controller with an Animation.

To animate a Particle System property, open the **Animation Window** with the GameObject containing the Animator and Particle System selected. Click **Add Property** to add properties.



Add properties to the animation in the Animation Window.

Scroll to the right to reveal the **add controls**.



Note that for curves, you can only keyframe the overall **curve multiplier**, which can be found next to the curve editor in the **Inspector**.

2017–09–19 Page amended with limited editorial review

GameObject menu changed in Unity 4.6

Particle System How-Tos

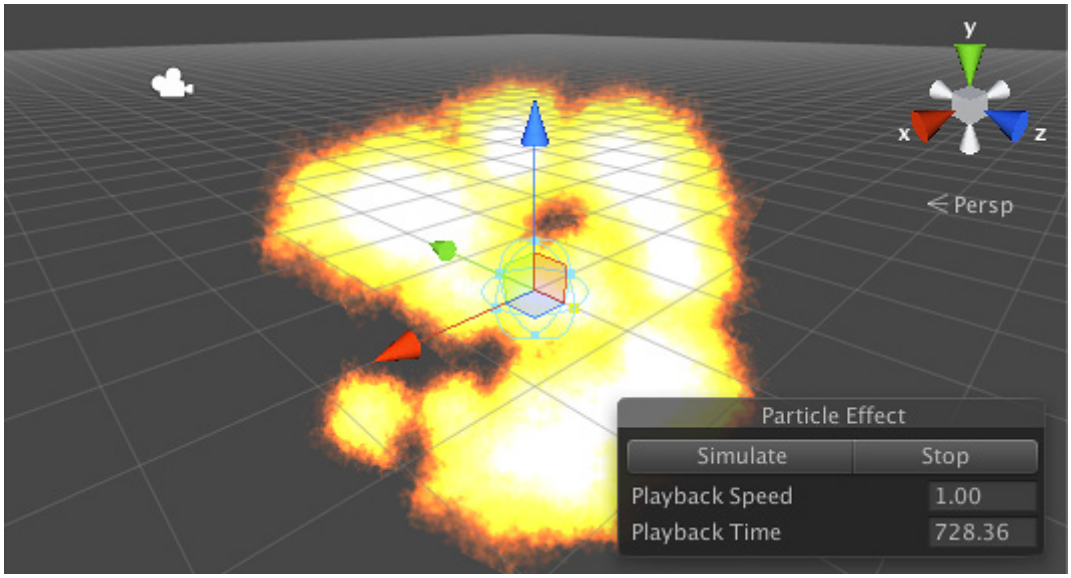
Leave feedback

This section explains how to implement common types of **particle system**. As with all code in our documentation, you are free to use it for any purpose without crediting Unity.

A Simple Explosion

Leave feedback

You can use a **particle system** to create a convincing explosion but the dynamics are perhaps a little more complicated than they seem at first. At its core, an explosion is just an outward burst of particles but there are a few simple modifications you can apply to make it look much more realistic.



A particle system explosion during development

Timeline of a Particle

A simple explosion produces a ball of flame that expands outward rapidly in all directions. The initial burst has a lot of energy and is therefore very hot (ie, bright) and moves very fast. This energy quickly dissipates which results in the expansion of flame slowing down and also cooling down (ie, getting less bright). Finally, as all the fuel is burned up, the flames will die away and soon disappear completely.

An explosion particle will typically have a short lifetime and you can vary several different properties over that lifetime to simulate the effect. The particle will start off moving very fast but then its speed should reduce greatly as it moves away from the centre of the explosion. Also, the color should start off bright but then darken and eventually fade to transparency. Finally, reducing the particle's size over its lifetime will give the effect of the flames dispersing as the fuel is used up.

Implementation

Starting with the default particle system object (menu: **GameObject > Effects > Particle System**), go to the *Shape* module and set the emitter shape to a small *Sphere*, say about 0.5 units in radius. The particles in the standard assets include a material called **ParticleFireball** which is very suitable for explosions (menu: **Assets > Import Package > ParticleSystems**). You can set this material for the system using the *Renderer* module. With the *Renderer* open, you should also disable *Cast Shadows* and *Receive Shadows* since the explosion flames are supposed to give out light rather than receive it.

At this stage, the system looks like lots of little fireballs being thrown out from a central point. The explosion should, of course, create a burst with lots of particles all at once. In the *Emission* module, you can set the *Rate* value to zero and add a single *Burst* of particles at time zero. The number of particles in the burst will depend on the size and intensity you want your explosion to have but a good starting point is about fifty particles. With the burst set up, the system is now starting to look much more like an explosion, but it is rather slow and the flames

seem to hang around for a long time. In the Particle System module (which will have the same name as the GameObject, eg, “Explosion”), set both the *Duration* of the system and the *Start Lifetime* of the particles to two seconds.

You can also use the *Size Over Lifetime* module to create the effect of the flames using up their fuel. Set the size curve using the “ramp down” preset (ie, the size starts off at 100% and reduces to zero. To make the flames darken and fade, enable the *Color Over Lifetime* module and set the gradient to start with white at the left and finish with black at the right. The *Fire Add* material uses an additive shader for rendering, so the darkness of the color property also controls the transparency of the particle; the flames will become fully transparent as the color fades to black. Also, the additive material allows the brightness of particles to “add” together as they are drawn on top of each other. This helps to further enhance the impression of a bright flash at the start of the explosion when the particles are all close together.

As it stands, the explosion is taking shape but it looks as though it is happening out in space. The particles get thrown out and travel a long distance at constant speed before fading. If your game is set in space then this might be the exact effect you want. However, an explosion that happens in the atmosphere will be slowed and dampened by the surrounding air. Enable the *Limit Velocity Over Lifetime* module and set the *Speed* to about 3.0 and the *Dampen* fraction to about 0.4 and you should see the explosion lose a little strength as it progresses.

A final thing to note is that as the particles move away from the centre of the explosion, their individual shapes become more recognisable. In particular, seeing the particles all at the same size and with the same rotation makes it obvious that the same graphic is being reused for each particle. A simple way to avoid this is to add a bit of random variation to the size and rotation of the particles as they are generated. In the Particle System module at the top of the **inspector**, click the small arrow to the right of the *Start Size* and *Start Rotation* properties and set them both to *Random Between Two Constants*. For the rotation, set the two values to 0 and 360 (ie, completely random rotation). For the size, set the values to 0.5 and 1.5 to give some variation without the risk of having too many huge or tiny particles. You should now see that the repetition of particle graphics is now much less noticeable.

Usage

During testing, it is useful to have the *Looping* property switched on so you can see the explosion repeatedly but in the finished game, you should switch this off so the explosion happens only once. When the explosion is designed for an object that has the potential to explode (a fuel tank, say) you might want to add the Particle System **component** to the object with the *Play On Awake* property disabled. You can then set off the explosion from a script as necessary.

```
void Explode() {
    var exp = GetComponent<ParticleSystem>();
    exp.Play();
    Destroy(gameObject, exp.duration);
}
```

In other cases, explosions happen at points of impact. If the explosion originates from an object (eg, a grenade) then you could call the `Explode` function detailed above after a time delay or when it makes contact with the target.

```
// Grenade explodes after a time delay.
public float fuseTime;

void Start() {
    Invoke("Explode", fuseTime);
}



// Grenade explodes on impact.
void OnCollisionEnter(Collision coll) {
    Explode();
}
```

Where the explosion comes from an object that is not actually represented in the game (eg, a projectile that travels too fast to be seen), you can just instantiate an explosion in the appropriate place. You might determine the contact point from a raycast, for example.

```
// On the explosion object.
void Start() {
    var exp = GetComponent<ParticleSystem>();
    exp.Play();
    Destroy(gameObject, exp.duration);
}


// Possible projectile script.
public GameObject explosionPrefab;

void Update() {
    RaycastHit hit;

    if (Physics.Raycast (Camera.main.ScreenPointToRay (Input.mousePosition),
        Instantiate (explosionPrefab, hit.point, Quaternion.identity);
    }
}
```

Further Ideas

The explosion developed here is very basic but you can modify various aspects of it to get the exact feel you are looking for in your game.

The particle graphic you use will have a big effect on how the player “reads” the explosion. Having lots of small, separately recognisable flames suggests burning pieces being thrown out. Larger particles that don’t move completely apart appear more like a fireball fed by a destroyed fuel tank. Typically, you will need to change several properties together to complete the effect. For example, the fireball will persist longer and expand less before it disappears while a sharp burst may scatter burning pieces quite some distance.

A few properties are set with random values here but other many properties have a *Random Between Two Constants/Curves* option and you can use these to add variation in all sorts of ways. Varying the size and rotation helps to avoid the most obvious effects of particle repetition but you might also consider adding some randomness to the *Start Delay*, *Start Lifetime* and *Start Speed* properties. A small amount of variation helps to reinforce the impression of the explosion being a “natural” and unpredictable effect rather than a controlled mechanical process. Larger variations suggest a “dirty” explosion. For example, varying the *Start Delay* will produce an explosion that is no longer sharp but bursts more slowly, perhaps because fuel tanks in a vehicle are being separately ignited.

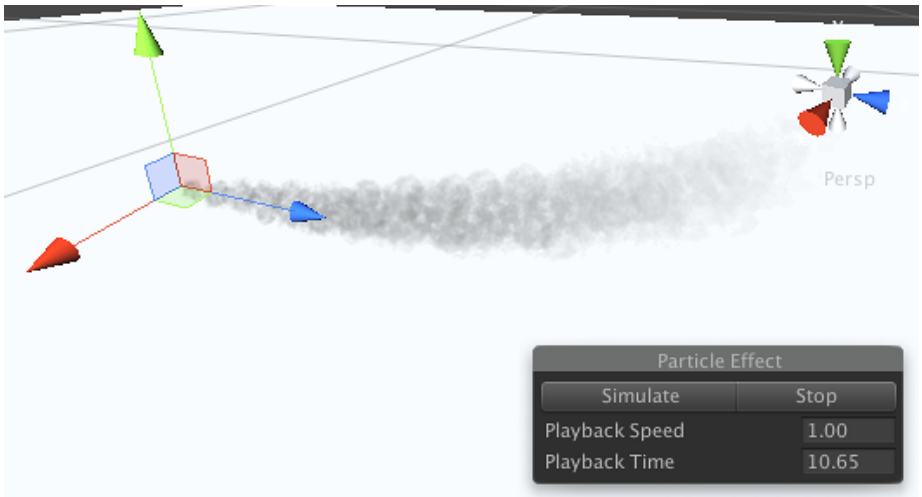
2017–09–19 Page amended with limited editorial review

GameObject menu changed in Unity 4.6

Exhaust Smoke from a Vehicle

Leave feedback

Cars and many other vehicles emit exhaust smoke as they convert fuel into power. You can use a **particle system** to add an exhaust as a nice finishing touch for a vehicle.



An exhaust generated by a particle system

Timeline of a Particle

Exhaust smoke emerges from the pipe quite fast but then rapidly slows down on contact with the atmosphere. As it slows, it spreads out, becoming fainter and soon dissipating into the air. Since the exhaust gas is hot, it also rises slightly as it passes through the colder air surrounding it.

A particle of exhaust smoke must start off no larger than the width of the pipe but it will then grow in size considerably over its short lifetime. It will usually start off partly transparent and fade to total transparency as it mixes with the air. As regards dynamics, the particle will be emitted quite fast but then slow rapidly and will also lift upward slightly.

Implementation

In the *Shape* module, select the *Cone* shape and set its *Angle* property to zero; the "cone" in this case will actually be a cylindrical pipe. The *Radius* of the pipe naturally depends on the size of the vehicle but you can usually set it by matching the radius **Gizmo** in the **scene view** to the vehicle model (eg, a car model will usually feature an exhaust pipe or a hole at the back whose size you can match). The radius actually determines quite a few things about the property settings you choose, such as the particle size and emission rate. For the purposes of this example, we will assume the vehicle is a car which follows Unity's standard size convention of one world unit to one metre; the radius is thus set to about 0.05 or 5cm.

A suitable graphic for the smoke particle is provided by the *Smoke4* material provided in the standard assets. If you don't already have these installed then select **Assets > Import Package > Particles** from the menu. Then, go to the *Renderer* module of the particle system and set the *Material* property to *Smoke4*.

The default lifetime of five seconds is generally too long for car exhaust fumes, so you should open the Particle System module (which has the same name as the GameObject, eg, "Exhaust") and set the *Start Lifetime* to about 2.5 seconds. Also in this module, set the *Simulation Space* to *World* and the *Gravity Modifier* to a small negative value, say about –0.1. Using a world simulation space allows the smoke to hang where it is produced even when the vehicle moves. The negative gravity effect causes the smoke particles to rise as if they are composed of hot gas. A nice extra touch is to use the small menu arrow next to *Start Rotation* to select the *Random Between Two Constants* option. Set the two values to 0 and 360, respectively, and the smoke particles will be randomly rotated as they are emitted. Having many particles that are identically aligned is very noticeable and detracts from the effect of a random, shapeless smoke trail.

At this stage, the smoke particles are starting to look realistic and the default emission rate creates a nice "chugging" effect of an engine. However, the smoke doesn't billow outwards and dissipate as yet. Open the *Color Over Lifetime* module and click the top gradient stop on the right hand end of the gradient (this controls the transparency of "alpha" value of the color). Set the alpha value

to zero and you should see the smoke particles fading to nothing in the **scene**. Depending on how clean your engine is, you may also want to reduce the alpha value of the gradient at the start; thick, dark smoke tends to suggest dirty, inefficient combustion.

As well as fading, the smoke should also increase in size as it escapes and you can easily create this effect with the *Size Over Lifetime* module. Open the module, select the curve and slide the curve handle at the left hand end to make the particles start off at a fraction of their full size. The exact size you choose depends on the size of the exhaust pipe but a value slightly larger than the pipe gives a good impression of escaping gas. (Starting the particles at the same size as the pipe suggests that the gas is being held to its shape by the pipe but of course, gas doesn't have a defined shape.) Use the simulation of the particle system in the scene view to get a good visual impression of how the smoke looks. You may also want to increase the *Start Size* in the particle system module at this point if the smoke doesn't disperse far enough to create the effect you want.

Finally, the smoke should also slow down as it disperses. An easy way to make this happen is with the *Force Over Lifetime* module. Open the module and set the *Space* option to *Local* and the Z **component** of the force to a negative value to indicate that the particles are pushed back by the force (the system emits the particles along the positive Z direction in the object's local space). A value of about –0.75 works quite well for the system if the other parameters are set up as suggested above.

Usage

You can position the exhaust particle system by placing it on a child object of the main vehicle. For simple games, you can just enable the *Play On Awake* and *Looping* properties and let the system run. In most cases, however, you will probably want to vary at least the emission rate as the vehicle moves. This is firstly for authenticity (ie, an engine produces more smoke as it works harder) but it also helps to prevent the smoke particles from being spread out as the vehicle moves. A fast-moving vehicle with too low an emission rate will appear to produce distinct "puffs" of smoke, which is highly unrealistic.

You can vary the emission rate very easily from a script. If you have a variable in the script that represents the engine revs or the speed of the vehicle then you can simply multiply this value by a constant and assign the result to the ParticleSystem's emissionRate property.

```
// C#

using UnityEngine;
using System.Collections;

public class PartScriptTestCS : MonoBehaviour {

    public float engineRevs;
    public float exhaustRate;

    ParticleSystem exhaust;


    void Start () {
        exhaust = GetComponent<ParticleSystem>();
    }


    void Update () {
        exhaust.emissionRate = engineRevs * exhaustRate;
    }

}
```

```
// JS

var engineRevs: float;
var exhaustRate: float;

var exhaust: ParticleSystem;


function Start() {
    exhaust = GetComponent.<ParticleSystem>();
}


function Update () {
    exhaust.emissionRate = engineRevs * exhaustRate;
}
```

Further Ideas

The basic scheme creates quite a convincing impression of exhaust smoke but you will probably have noticed that the “character” of the engine changes as you vary the parameters. An poorly tuned, inefficient engine tends to burn its fuel incompletely, resulting in heavy, dark smoke that persists for a long time in the air. This would be perfect for an old farm tractor but not for a high-performance sports car. For a “clean” engine, you should use small values for the lifetime, opacity and size increase of the particles. For a “dirty” engine, you should increase these values and perhaps use the *Bursts* property of the *Emission* module to create the impression of the engine spluttering.

Particle System vertex streams and Standard Shader support

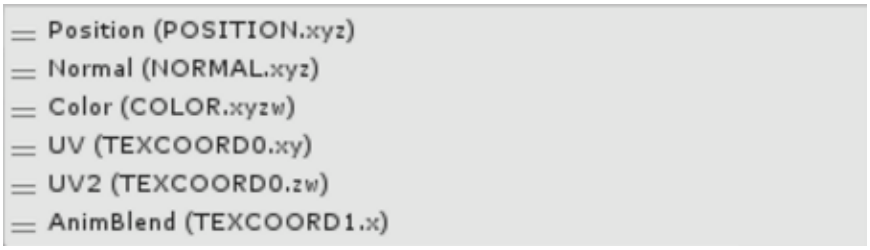
Leave feedback

If you are comfortable writing your own **Shaders**, use this addition to the Renderer Module to configure your **Particle Systems** to pass a wider range of data into your custom Shaders.

There are a number of built-in data streams to choose from, such as velocity, size and center position. Aside from the ability to create powerful custom Shaders, these streams allow a number of more general benefits:

Use the Tangent stream to support normal mapped particles.
You can remove Color and then add the Tangent UV2 and AnimBlend streams to use the **Standard Shader** on particles.
To easily perform linear texture blending of flipbooks, add the UV2 and AnimBlend streams, and attach the Particles/Anim Alpha Blended Shader (see example screenshot below to see how to set this up).
There are also two completely custom per-particle data streams (ParticleSystemVertexStreams.Custom1 and ParticleSystemVertexStreams.Custom2), which can be populated from script. Call SetCustomParticleData and GetCustomParticleData with your array of data to use them. There are two ways of using this:

To drive custom behavior in **scripts** by attaching your own data to particles; for example, attaching a "health" value to each particle.
To pass this data into a Shader by adding one of the two custom streams, in the same way you would send any other stream to your Shader (see ParticleSystemRenderer.EnableVertexStreams). To elaborate on the first example, maybe your custom health attribute could now also drive some kind of visual effect, as well as driving script-based game logic.
When adding vertex streams, Unity will provide you with some information in brackets, next to each item, to help you read the correct data in your shader:

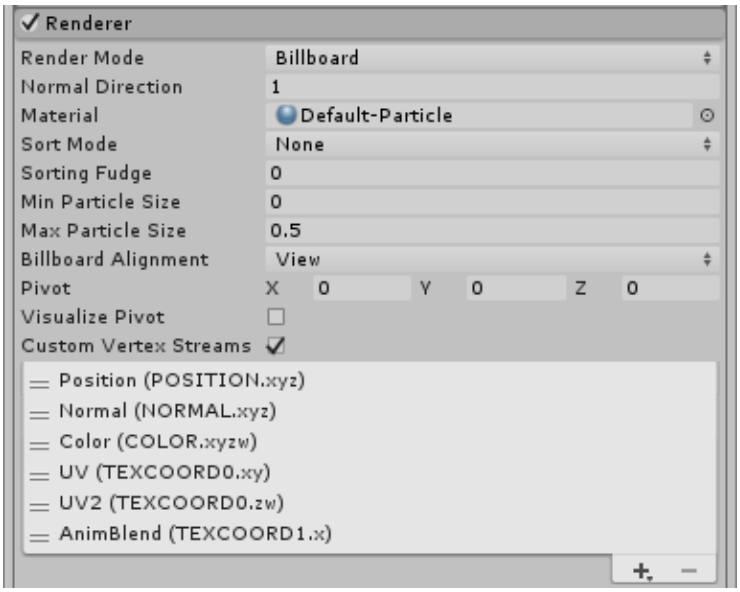


Each item in brackets corresponds to a **Vertex Shader** input, which you should specify in your Shader. Here is the correct input structure for this configuration.

```
struct appdata_t {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    fixed4 color : COLOR;
    float4 texcoords : TEXCOORD0;
    float texcoordBlend : TEXCOORD1;
};
```

Notice that both UV and UV2 are passed in different parts of TEXCOORD0, so we use a single declaration for both. To access each one in your shader, you would use the xy and zw swizzles. This allows you to pack your vertex data efficiently.

Here is an example of an animated flip-book Shader. It uses the default inputs (**Position**, **Normal**, **Color**, **UV**), but also uses two additional streams for the second UV stream (**UV2**) and the flip-book frame information (**AnimBlend**).




```
Shader "Particles/Anim Alpha Blended" {
Properties {
    _TintColor ("Tint Color", Color) = (0.5,0.5,0.5,0.5)
    _MainTex ("Particle Texture", 2D) = "white" {}
    _InvFade ("Soft Particles Factor", Range(0.01,3.0)) = 1.0
}

Category {
    Tags { "Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"
    Blend SrcAlpha OneMinusSrcAlpha
    ColorMask RGB
    Cull Off Lighting Off ZWrite Off

    SubShader {
        Pass {

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma target 2.0
            #pragma multi_compile_particles
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            sampler2D _MainTex;
```

```
fixed4 _TintColor;

struct appdata_t {
    float4 vertex : POSITION;
    fixed4 color : COLOR;
    float4 texcoords : TEXCOORD0;
    float texcoordBlend : TEXCOORD1;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct v2f {
    float4 vertex : SV_POSITION;
    fixed4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    float2 texcoord2 : TEXCOORD1;
    fixed blend : TEXCOORD2;
    UNITY_FOG_COORDS(3)
    #ifdef SOFTPARTICLES_ON
    float4 projPos : TEXCOORD4;
    #endif
    UNITY_VERTEX_OUTPUT_STEREO
};

float4 _MainTex_ST;

v2f vert (appdata_t v)
{
    v2f o;
    UNITY_SETUP_INSTANCE_ID(v);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);
    o.vertex = UnityObjectToClipPos(v.vertex);
    #ifdef SOFTPARTICLES_ON
    o.projPos = ComputeScreenPos (o.vertex);
    COMPUTE_EYEDEPTH(o.projPos.z);
    #endif
    o.color = v.color * _TintColor;
    o.texcoord = TRANSFORM_TEX(v.texcoords.xy,_MainTex);
    o.texcoord2 = TRANSFORM_TEX(v.texcoords.zw,_MainTex);
    o.blend = v.texcoordBlend;
    UNITY_TRANSFER_FOG(o,o.vertex);
    return o;
}

sampler2D_float _CameraDepthTexture;
float _InvFade;

fixed4 frag (v2f i) : SV_Target
{
```

```
                #ifdef SOFTPARTICLES_ON
                float sceneZ = LinearEyeDepth (SAMPLE_DEPTH_TEXTURE_PROJ(_CameraDept
                float partZ = i.projPos.z;
                float fade = saturate (_InvFade * (sceneZ-partZ));
                i.color.a *= fade;
                #endif

                fixed4 colA = tex2D(_MainTex, i.texcoord);
                fixed4 colB = tex2D(_MainTex, i.texcoord2);
                fixed4 col = 2.0f * i.color * lerp(colA, colB, i.blend);
                UNITY_APPLY_FOG(i.fogCoord, col);
                return col;
            }
            ENDCG
        }
    }
}
}
```

It's also possible to use **Surface Shaders** with this system, although there are some extra things to be aware of:

The input structure to your surface function is not the same as the input structure to the vertex Shader. You need to provide your own vertex Shader input structure. See below for an example, where it is called `appdata_particles`.
When surface Shaders are built, there is automatic handling of variables whose names begin with certain tokens. The most notable one is `uv`. To prevent the automatic handling from causing problems here, be sure to give your UV inputs different names (for example, "texcoord").
Here is the same functionality as the first example, but in a Surface Shader:

```
Shader "Particles/Anim Alpha Blend Surface" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader {
        Tags {"Queue"="Transparent" "RenderType"="Transparent"}
        Blend SrcAlpha OneMinusSrcAlpha
        ZWrite off
        LOD 200

        CGPROGRAM
```

```
        // Physically based Standard lighting model, and enable shadows on all light
        #pragma surface surf Standard alpha vertex:vert

        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;

         struct appdata_particles {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float4 color : COLOR;
            float4 texcoords : TEXCOORD0;
            float texcoordBlend : TEXCOORD1;
            };


        struct Input {
            float2 uv_MainTex;
            float2 texcoord1;
            float blend;
            float4 color;
        };


        void vert(inout appdata_particles v, out Input o) {
            UNITY_INITIALIZE_OUTPUT(Input,o);
            o.uv_MainTex = v.texcoords.xy;
            o.texcoord1 = v.texcoords.zw;
            o.blend = v.texcoordBlend;
            o.color = v.color;
          }


        half _Glossiness;
        half _Metallic;
        fixed4 _Color;


        void surf (Input IN, inout SurfaceOutputStandard o) {
            fixed4 colA = tex2D(_MainTex, IN.uv_MainTex);
            fixed4 colB = tex2D(_MainTex, IN.texcoord1);
            fixed4 c = 2.0f * IN.color * lerp(colA, colB, IN.blend) * _Color;

            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
```

```
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

2017–05–15 Page amended with no editorial review

Particle System GPU Instancing

Leave feedback

GPU instancing offers a large performance boost compared with CPU **rendering**. You can use it if you want your **particle system** to render **Mesh** particles (as opposed to the default rendering mode of rendering **billboard** particles).

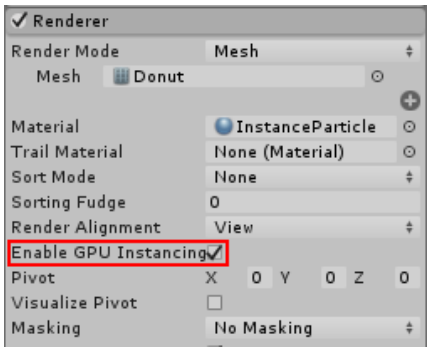
To be able to use GPU instancing with your particle systems:

Set your Particle System's renderer mode to **Mesh**

Use a **shader** for the renderer material that supports GPU Instancing

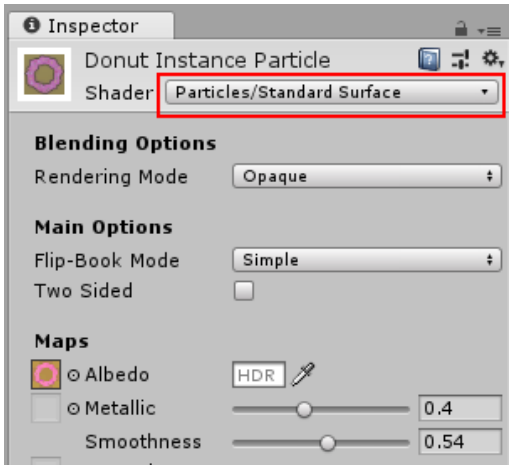
Run your project on a platform that supports GPU instancing

To enable GPU instancing for a particle system, you must enable the **Enable GPU Instancing** checkbox in the **Renderer** module of your particle system.



The option to enable Particle System GPU instancing in the Renderer module

Unity comes with a built-in particle shader that supports GPU instancing, but the default particle material does not use it, so you must change this to use GPU instancing. The particle shader that supports GPU instancing is called **Particles/Standard Surface**. To use it, you must create your own new **material**, and set the material's shader to **Particles/Standard Surface**. You must then assign this new material to the material field in the Particle System renderer module.



The built-in shader that is compatible with Particle System GPU Instancing

If you are using a different shader for your particles, it must use '#pragma target 4.5' or higher. See Shader Compile Targets for more details. This requirement is higher than regular GPU Instancing in Unity because the Particle System writes all its instance data to a single large buffer, rather than breaking up the instancing into multiple draw calls.

Custom shader examples

You can also write custom shaders that make use of GPU Instancing. See the following sections for more information:

Particle system GPU Instancing in a Surface Shader
Particle system GPU Instancing in a Custom Shader
Customising instance data used by the Particle System (to work alongside Custom Vertex Streams)

Particle system GPU Instancing in a Surface Shader

Here is a complete working example of Surface Shader using Particle System GPU Instancing:

```
Shader "Instanced/ParticleMeshesSurface" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light types
        // And generate the shadow pass with instancing support
        #pragma surface surf Standard nolightmap nometa noforwardadd keepalpha fullforwardshadow
        // Enable instancing for this shader
        #pragma multi_compile_instancing
        #pragma instancing_options procedural:vertInstancingSetup
        #pragma exclude_renderers gles
        #include "UnityStandardParticleInstancing.cginc"
        sampler2D _MainTex;
        struct Input {
            float2 uv_MainTex;
            fixed4 vertexColor;
        };
        fixed4 _Color;
        half _Glossiness;
        half _Metallic;
        void vert (inout appdata_full v, out Input o)
        {
            UNITY_INITIALIZE_OUTPUT(Input, o);
            vertInstancingColor(o.vertexColor);
            vertInstancingUVs(v.texcoord, o.uv_MainTex);
        }

        void surf (Input IN, inout SurfaceOutputStandard o) {
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * IN.vertexColor * _Color;
            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

There are a number of small differences to a regular Surface Shader in the above example, which make it work with particle instancing.

Firstly, you must add the following two lines to enable Procedural Instancing, and specify the built-in vertex setup function. This function lives in UnityStandardParticleInstancing.cginc, and loads the per-instance (per-particle) positional data:

```
        #pragma instancing_options procedural:vertInstancingSetup
        #include "UnityStandardParticleInstancing.cginc"
```

The other modification in the example is to the Vertex function, which has two extra lines that apply per-instance attributes, specifically, particle colors and Texture Sheet Animation texture coordinates:

```
            vertInstancingColor(o.vertexColor);
            vertInstancingUVs(v.texcoord, o.uv_MainTex);
```

Particle System GPU Instancing in a Custom Shader

Here is a complete working example of a Custom Shader using particle system GPU instancing. This custom shader adds a feature which the standard particle shader does not have - a fade between the individual frames of a texture sheet animation.

```
Shader "Instanced/ParticleMeshesCustom"
{
    Properties
    {
        _MainTex("Albedo", 2D) = "white" {}
        [Toggle(_TSANIM_BLENDING)] _TSAnimBlending("Texture Sheet Animation Blending", Int) = 0
    }
    SubShader
    {
        Tags{ "RenderType" = "Opaque" }
        LOD 100
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile __ _TSANIM_BLENDING
            #pragma multi_compile_instancing
            #pragma instancing_options procedural:vertInstancingSetup
            #include "UnityCG.cginc"
            #include "UnityStandardParticleInstancing.cginc"
            struct appdata
            {
                float4 vertex : POSITION;
                fixed4 color : COLOR;
                float2 texcoord : TEXCOORD0;
                UNITY_VERTEX_INPUT_INSTANCE_ID
            };
            struct v2f
            {
```

```
                float4 vertex : SV_POSITION;
                fixed4 color : COLOR;
                float2 texcoord : TEXCOORD0;
#ifdef _TSANIM_BLENDING
                float3 texcoord2AndBlend : TEXCOORD1;
#endif
            };
            sampler2D _MainTex;
            float4 _MainTex_ST;
            fixed4 readTexture(sampler2D tex, v2f IN)
            {
                fixed4 color = tex2D(tex, IN.texcoord);
#ifdef _TSANIM_BLENDING
                fixed4 color2 = tex2D(tex, IN.texcoord2AndBlend.xy);
                color = lerp(color, color2, IN.texcoord2AndBlend.z);
#endif
                return color;
            }
            v2f vert(appdata v)
            {
                v2f o;
                UNITY_SETUP_INSTANCE_ID(v);
                o.color = v.color;
                o.texcoord = v.texcoord;
                vertInstancingColor(o.color);
#ifdef _TSANIM_BLENDING
                vertInstancingUVs(v.texcoord, o.texcoord, o.texcoord2AndBlend);
#else
                vertInstancingUVs(v.texcoord, o.texcoord);
#endif
                o.vertex = UnityObjectToClipPos(v.vertex);
                return o;
            }
            fixed4 frag(v2f i) : SV_Target
            {
                half4 albedo = readTexture(_MainTex, i);
                return i.color * albedo;
            }
            ENDCG
        }
    }
}
```

This example contains the same set-up code as the Surface Shader for loading the positional data:

```
        #pragma instancing_options procedural:vertInstancingSetup
        #include "UnityStandardParticleInstancing.cginc"
```

The modification to the vertex function is very similar to the Surface Shader too:

```
                vertInstancingColor(o.color);
#ifdef _TSANIM_BLENDING
                vertInstancingUVs(v.texcoord, o.texcoord, o.texcoord2AndBlend);
#else
                vertInstancingUVs(v.texcoord, o.texcoord);
#endif
```

The only difference here, compared with the first example above, is the texture sheet animation blending. This means that the shader requires an extra set of texture coordinates to read two frames of the texture sheet animation instead of just one, and blends them together.

Finally, the **fragment shader** reads the textures and calculates the final color.

Particle system GPU Instancing with custom vertex streams

The examples above only use the default vertex stream setup for particles. This includes a position, a normal, a color, and one UV. However, by using custom vertex streams, you can send other data to the shader, such as velocities, rotations and sizes.

In this next example, the shader is designed to display a special effect, which makes faster particles appear brighter, and slower particles dimmer. There is some extra code that brightens particles according to their speed, using the Speed Vertex Stream. Also, because this shader assumes the effect will not be using texture sheet animation, it is omitted from the custom stream struct.

Here is the full Shader:

```
Shader "Instanced/ParticleMeshesCustomStreams"
{
    Properties
    {
        _MainTex("Albedo", 2D) = "white" {}
    }
    SubShader
    {
        Tags{ "RenderType" = "Opaque" }
        LOD 100
        Pass
        {
            CGPROGRAM
#pragma exclude_renderers gles
            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile_instancing
            #pragma instancing_options procedural:vertInstancingSetup
            #define UNITY_PARTICLE_INSTANCE_DATA MyParticleInstanceData
            #define UNITY_PARTICLE_INSTANCE_DATA_NO_ANIM_FRAME
            struct MyParticleInstanceData
            {
                float3x4 transform;
                uint color;
                float speed;
            };
            #include "UnityCG.cginc"
            #include "UnityStandardParticleInstancing.cginc"
            struct appdata
```

```
            {
                float4 vertex : POSITION;
                fixed4 color : COLOR;
                float2 texcoord : TEXCOORD0;
                UNITY_VERTEX_INPUT_INSTANCE_ID
            };
            struct v2f
            {
                float4 vertex : SV_POSITION;
                fixed4 color : COLOR;
                float2 texcoord : TEXCOORD0;
            };
            sampler2D _MainTex;
            float4 _MainTex_ST;
            v2f vert(appdata v)
            {
                v2f o;
                UNITY_SETUP_INSTANCE_ID(v);
                o.color = v.color;
                o.texcoord = v.texcoord;
                vertInstancingColor(o.color);
                vertInstancingUVs(v.texcoord, o.texcoord);
#if defined(UNITY_PARTICLE_INSTANCING_ENABLED)
                UNITY_PARTICLE_INSTANCE_DATA data = unity_ParticleInstanceData[unity_InstanceID]
                o.color.rgb += data.speed;
#endif
                o.vertex = UnityObjectToClipPos(v.vertex);
                return o;
            }
            fixed4 frag(v2f i) : SV_Target
            {
                half4 albedo = tex2D(_MainTex, i.texcoord);
                return i.color * albedo;
            }
            ENDCG
        }
    }
}
```

The shader includes `UnityStandardParticleInstancing.cginc`, which contains the default instancing data layout for when Custom Vertex Streams are not being used. So, when using custom streams, you must override some of the defaults defined in that header. These overrides must come **before** the include. The example above sets the following custom overrides:

Firstly, there is a line that tells Unity to use a custom struct called 'MyParticleInstanceData' for the custom stream data, using the UNITY_PARTICLE_INSTANCE_DATA macro:

```
            #define UNITY_PARTICLE_INSTANCE_DATA MyParticleInstanceData
```

Next, another define tells the instancing system that the Anim Frame Stream is not required in this shader, because the effect in this example is not intended for use with texture sheet animation:

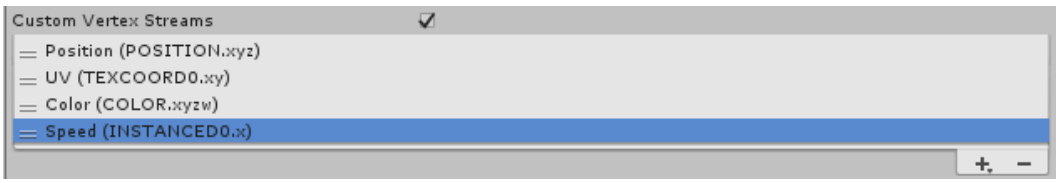
```
#define UNITY_PARTICLE_INSTANCE_DATA_NO_ANIM_FRAME
```

Thirdly, the struct for the custom stream data is declared:

```
struct MyParticleInstanceData
{
    float3x4 transform;
    uint color;
    float speed;
};
```

These overrides all come before `UnityStandardParticleInstancing.cginc` is included, so the shader does not use its own defaults for those defines.

When writing your struct, the variables must match the vertex streams listed in the **Inspector** in the Particle System renderer module. This means you must choose the streams you want to use in the Renderer module **UI**, and add them to variable definitions in your custom stream data struct in the same order, so that they match:



The custom vertex streams shown in the Renderer module UI, showing some instanced and some non-instanced streams

The first item (Position) is mandatory, so you cannot remove it. You can freely add/remove other entries using the plus and minus buttons to customize your vertex stream data.

Entries in the list that are followed by **INSTANCED** contain instance data, so you must include them in your particle instance data struct. The number directly appended to the word **INSTANCED** (for example zero in **INSTANCED0** and one in **INSTANCED1**) indicates the order in which the variables must appear in your struct, *after* the initial "transform" variable. The trailing letters (.x .xy .xyz or .xyzw) indicate the variable's type and map to float, float2, float3 and float4 variable types in your shader code.

You can omit any other vertex stream data that appears in the list, but that does *not* have the word **INSTANCED** after it, from the particle instance data struct, because it is not instanced data to be processed by the shader. This data belongs to the source mesh, for example UV's, Normals and Tangents.

The final step to complete our example is to apply the speed to the particle color inside the **Vertex Shader**:

```
#if defined(UNITY_PARTICLE_INSTANCING_ENABLED)
                UNITY_PARTICLE_INSTANCE_DATA data = unity_ParticleInstanceData[unity_InstanceID]
                o.color.rgb += data.speed;
#endif
```

You must wrap all the instancing code inside the check for UNITY_PARTICLE_INSTANCING_ENABLED, so that it can compile when instancing is not being used.

At this point, if you want to pass the data to the Fragment Shader instead, you can write the data into the v2f struct, like you would with any other shader data.

This example describes how to modify a Custom Shader for use with Custom Vertex Streams, but you can apply exactly the same approach to a Surface Shader to achieve the same functionality.

2018–03–28 Page published with editorial review

Particle System GPU instancing added in Unity 2018.1

Post-processing overview

Leave feedback

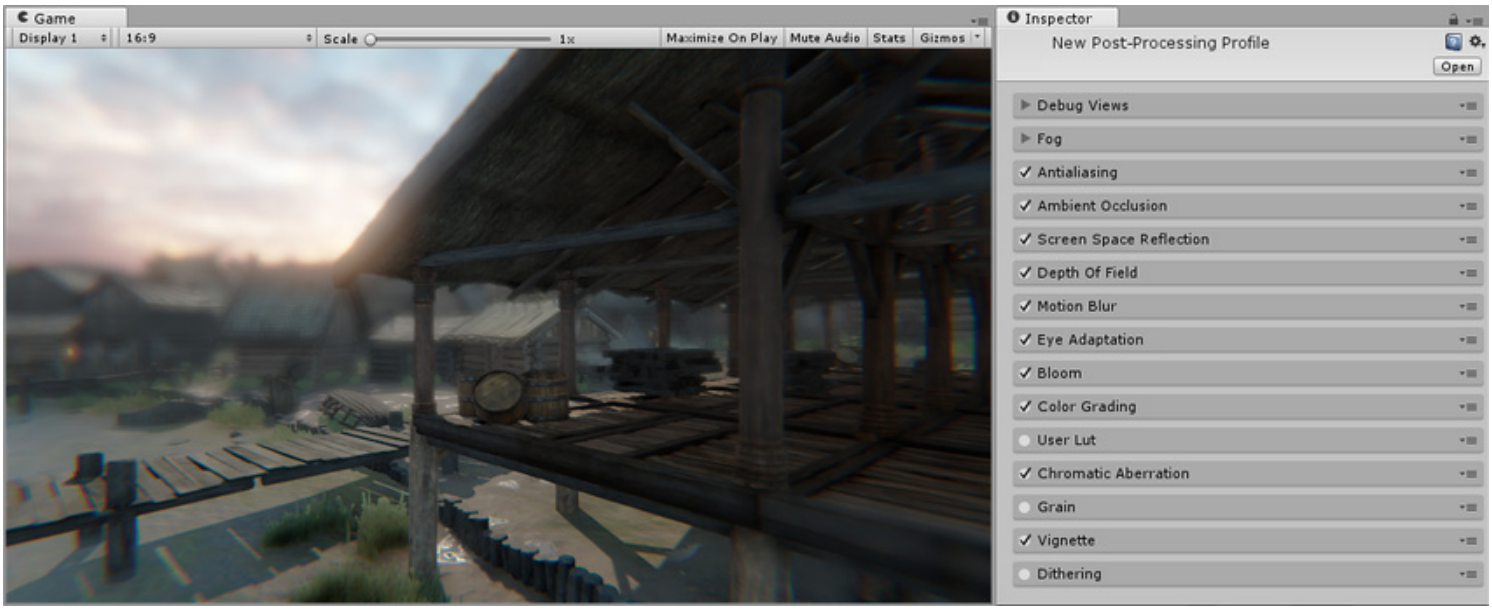
Post-processing is the process of applying full-screen filters and effects to a **camera**'s image buffer before it is displayed to screen. It can drastically improve the visuals of your product with little setup time.

You can use post-processing effects to simulate physical camera and film properties; for example; **Bloom**, **Depth of Field**, **Chromatic Aberration** or **Color Grading**.

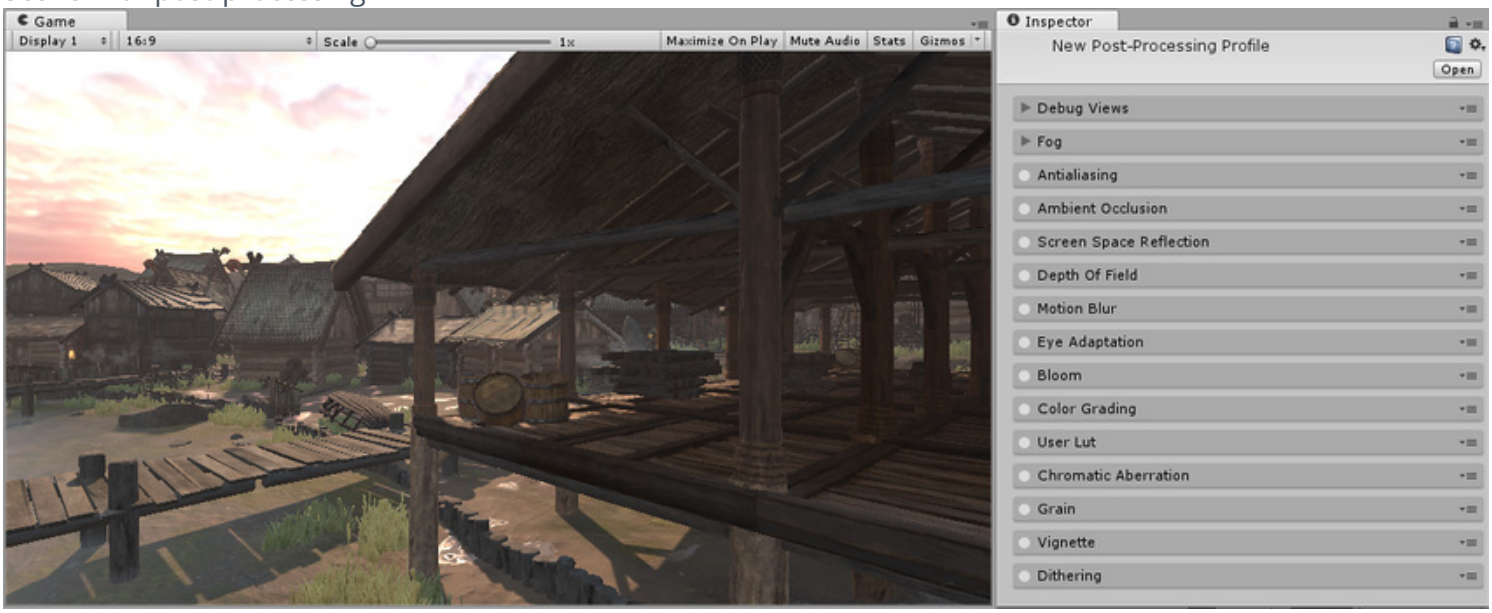
Using post-processing

To use post-processing in your project you can import Unity's post-processing stack. You can also write your own post-processing effects. See Writing post-processing effects for details.

The images below demonstrate a **Scene** with and without post-processing.



Scene with post-processing



Scene with no post-processing

2017–05–24 Page published with limited editorial review

New feature in 5.6

Post-processing stack

Leave feedback

The post-processing stack is an über effect that combines a complete set of effects into a single post-processing pipeline. This has a few advantages:

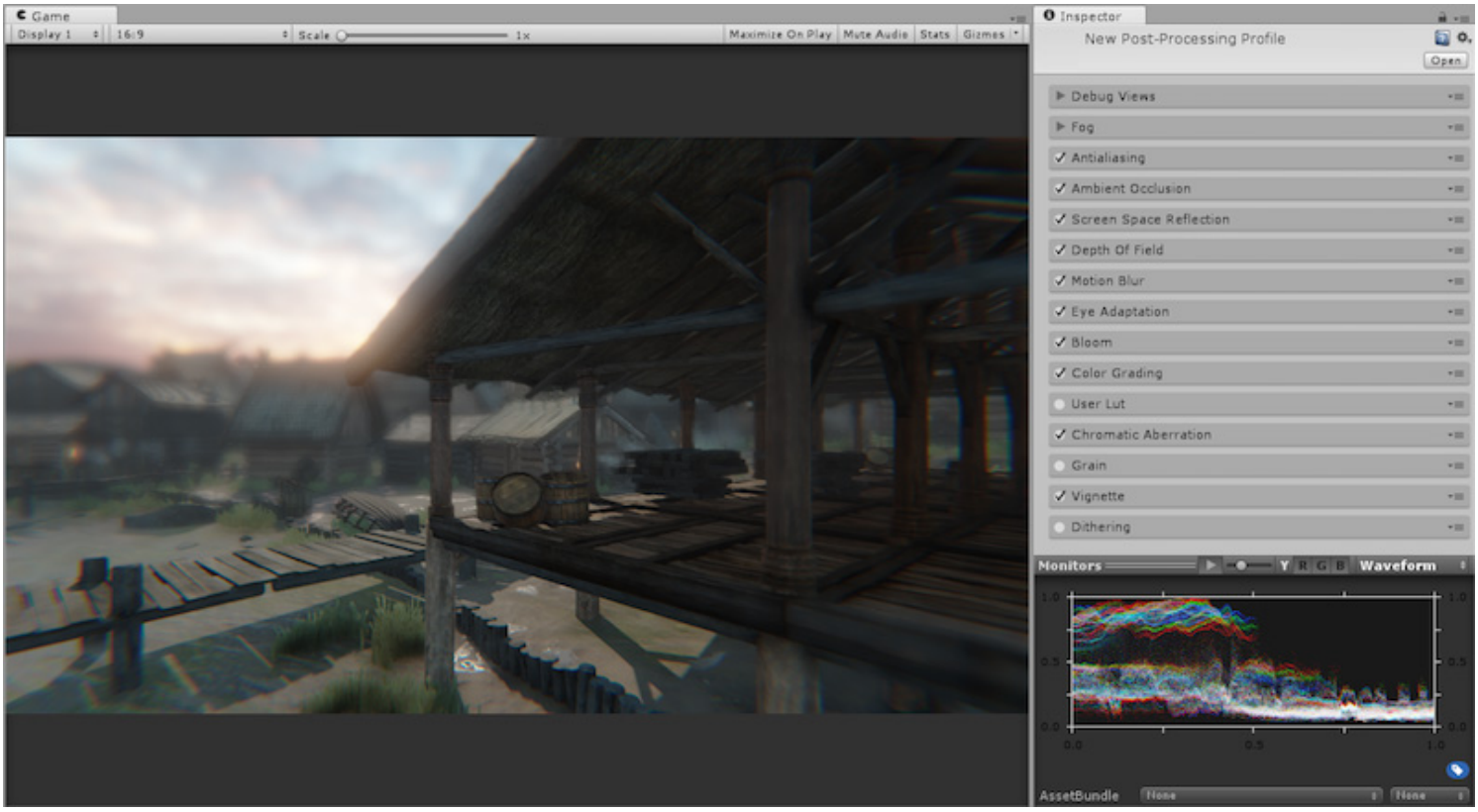
Effects are always configured in the correct order

It allows a combination of many effects into a single pass

All effects are grouped together in the **UI** for a better user experience

The post-processing stack also includes a collection of monitors and debug views to help you set up your effects correctly and debug problems in the output.

To use post-processing, download the post-processing stack from the Asset Store.



Post-processing stack

For help on how to get started with the post-processing stack, see Setting up the post-processing stack.

Effects

Anti-aliasing (FXAA & TAA)
Ambient Occlusion
Screen Space Reflection
Fog
Depth of Field
Motion Blur
Eye Adaptation
Bloom

Color Grading
User Lut
Chromatic Aberration
Grain
Vignette
Dithering

For details about each individual effect included in the stack see the page for that effect.

Post-processing stack version 2

For an early preview of the next version of the post processing stack, see Post-processing Stack v2 .

2017–09–04 Page amended with limited editorial review

New feature in 5.6

Added a link to the Post-processing Stack v2 GitHub branch, which is available as a preview in 2017.1

Setting up the post-processing stack

Leave feedback

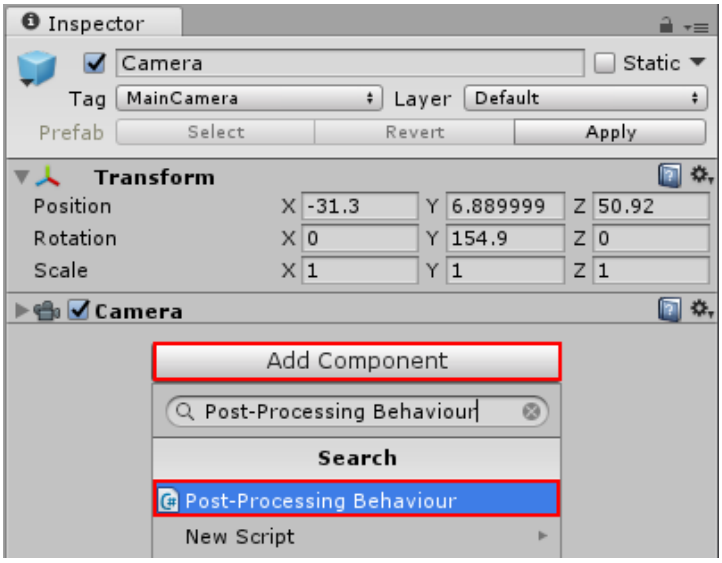
For optimal post-processing results it is recommended that you work in the Linear color-space with **HDR** enabled. Using the deferred **rendering path** is also recommended (as required for some effects, such as Screen Space Reflection).

First you need to add the Post Processing Behaviour script to your **camera**. You can do that by selecting your **camera** and use one of the following ways:

Drag the *PostProcessingBehaviour.cs* script from the project window to the camera.

Use the menu **Component** > **Effects** > **Post Processing Behaviour**.

Use the **Add Component** button in the **Inspector**.



Adding a Post-Processing Behaviour script

You will now have a behaviour configured with an empty profile. The next step is to create a custom profile using one of the following ways:

Right-click in your project window and select **Create** > **Post-Processing Profile**.

Use the menu **Assets** > **Create** > **Post-Processing Profile**.

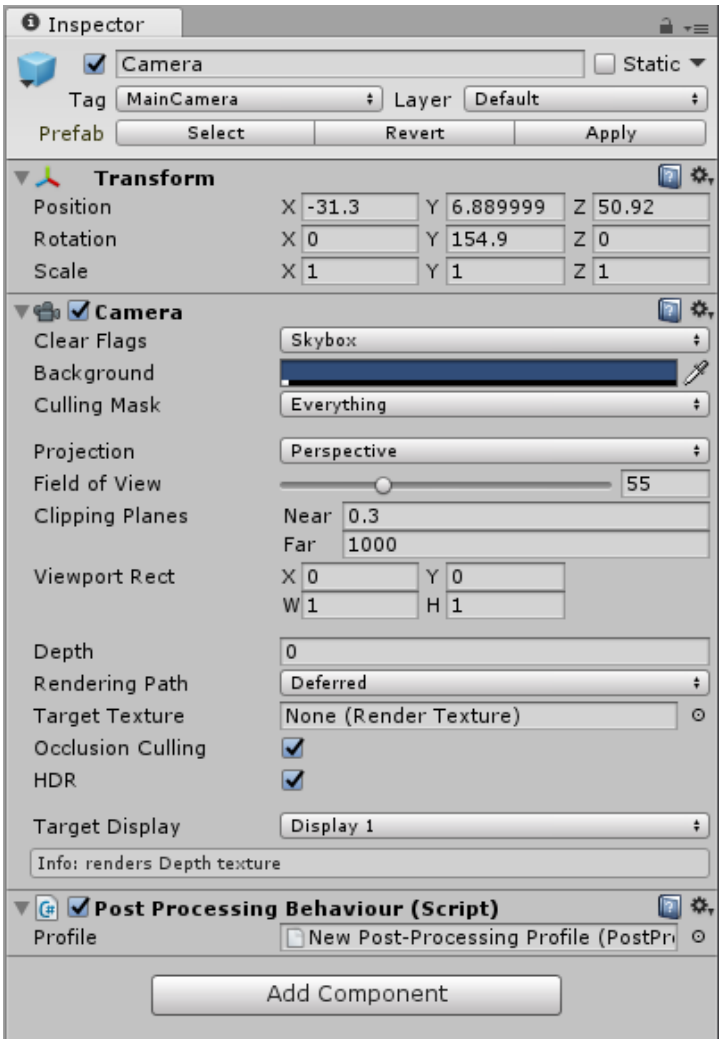
This will create a new asset in your project.

Post-Processing Profiles are project assets and can be shared easily between **scenes** / cameras, as well as between different projects or on the **Asset Store**. This makes creating presets easier (ie. high quality preset for desktop or lower settings for mobile).

Selecting a profile will show the inspector window for editing the profile settings.

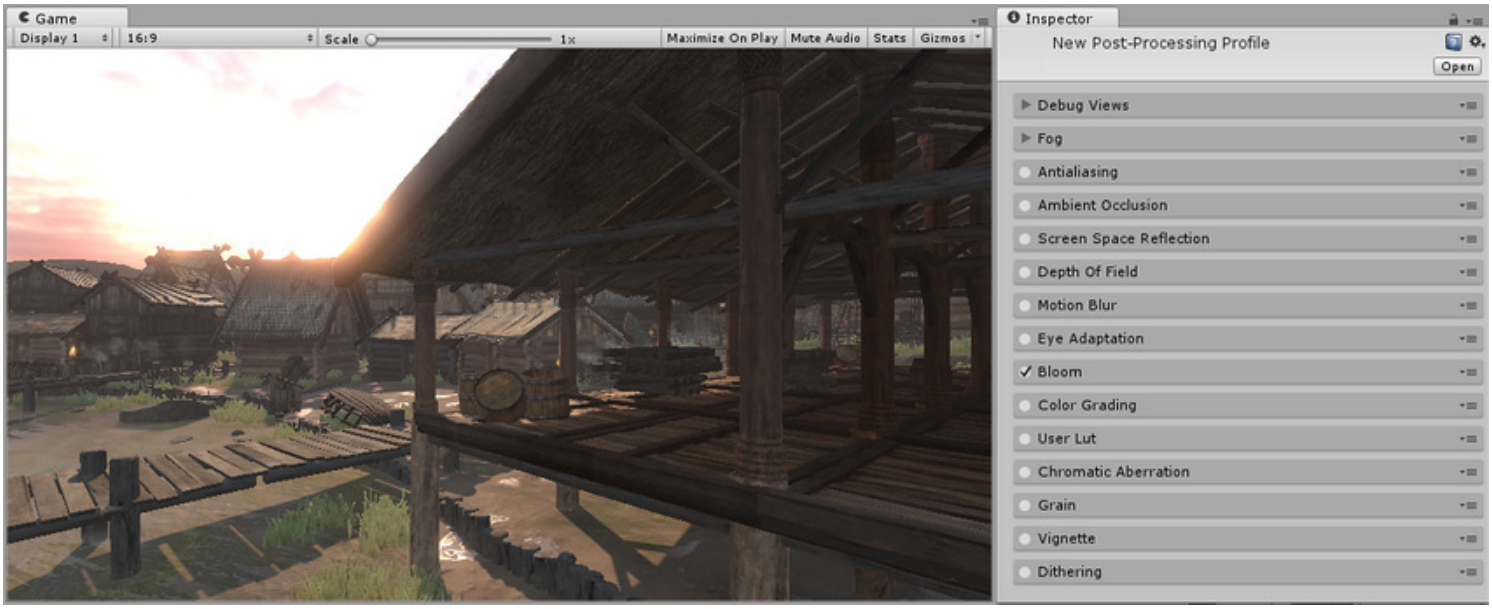
Newly created post-processing profile

To assign the profile to the behaviour you can drag it from the project panel to the component or use the object selector in the inspector.



Post-processing profile assigned to the Behaviour script

With the profile selected, you can use the checkbox on each effect in the inspector to enable or disable individual effects. You'll find more information about each effect in their individual documentation pages.



Bloom effect enabled

2017–05–24 Page published with limited editorial review

New feature in 5.6

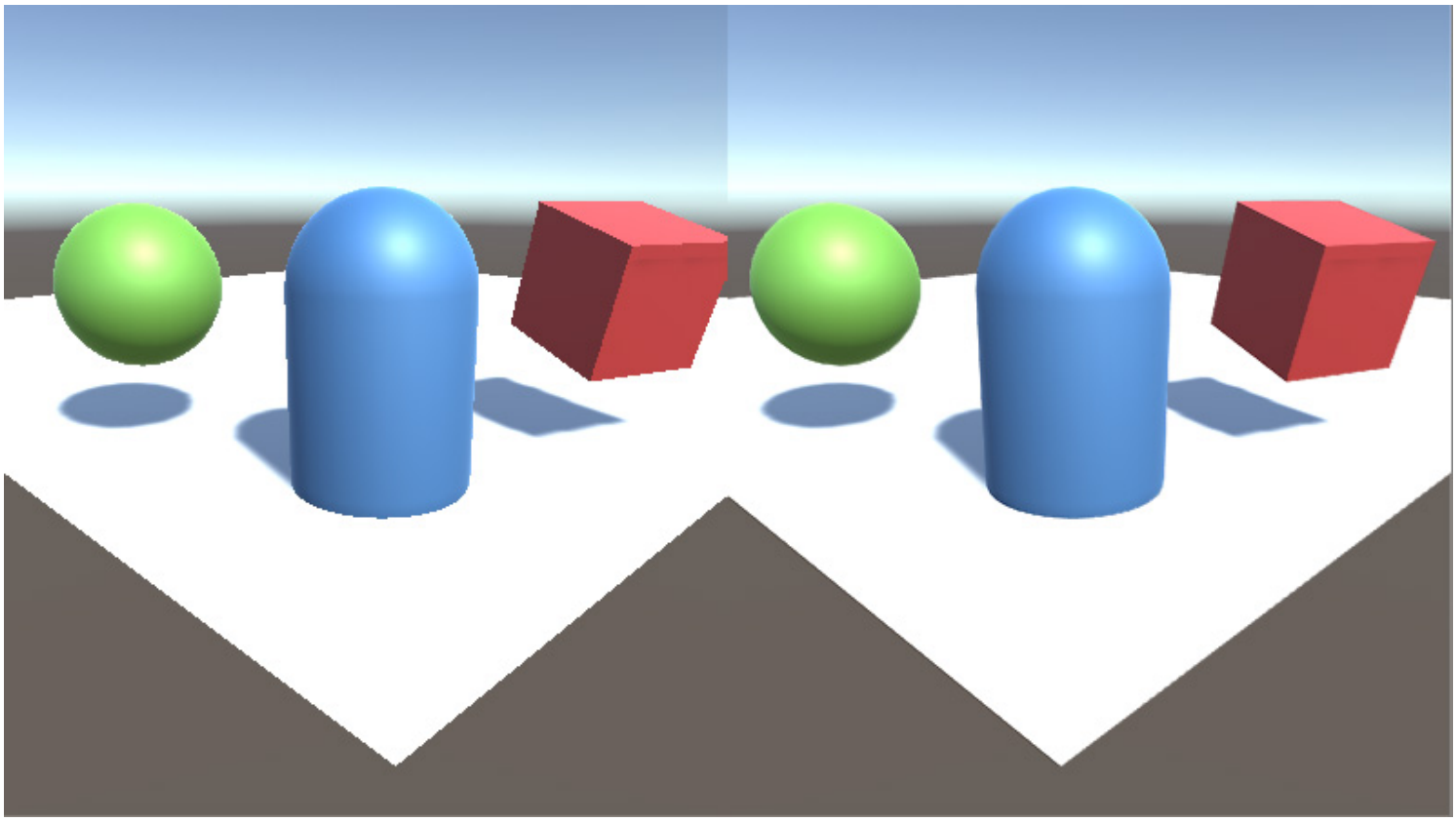
Anti-aliasing

Leave feedback

The effect descriptions on this page refer to the default effects found within the post-processing stack.

The **Anti-aliasing** effect offers a set of algorithms designed to prevent aliasing and give a smoother appearance to graphics. Aliasing is an effect where lines appear jagged or have a “staircase” appearance (as displayed in the left-hand image below). This can happen if the graphics output device does not have a high enough resolution to display a straight line.

Anti-aliasing reduces the prominence of these jagged lines by surrounding them with intermediate shades of color. Although this reduces the jagged appearance of the lines, it also makes them blurrier.



The Scene on the left is rendered without anti-aliasing. The Scene on the right shows the effect of the Temporal Anti-aliasing algorithm.

The Anti-aliasing algorithms are image-based. This is very useful when traditional multisampling (as used in the Editor’s Quality settings) is not properly supported, such as:

When using deferred rendering

When using **HDR** in the **forward rendering** path in Unity 5.5 or earlier

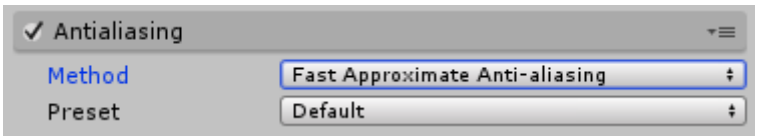
The algorithms supplied in the post-processing stack are:

Fast Approximate Anti-aliasing (FXAA)

Temporal Anti-aliasing (TAA)

Fast Approximate Anti-aliasing

FXAA is the cheapest technique and is recommended for mobile and other platforms that don't support motion vectors, which are required for TAA. However it contains multiple quality presets and as such is also suitable as a fallback solution for slower desktop and console hardware.



UI for the Anti-aliasing effect when FXAA is selected

Properties

Property:	Function:
Preset	The quality preset to be used. Provides a trade-off between performance and edge quality.

Optimisation

Reduce quality setting

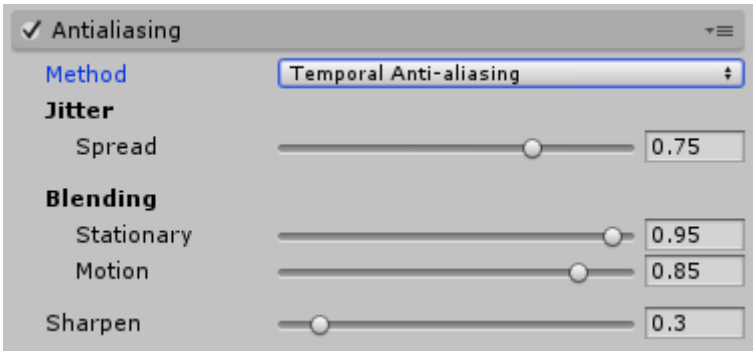
Requirements

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

Temporal Anti-aliasing

Temporal Anti-aliasing is a more advanced anti-aliasing technique where frames are accumulated over time in a history buffer to be used to smooth edges more effectively. It is substantially better at smoothing edges in motion but requires motion vectors and is more expensive than FXAA. Due to this it is recommended for desktop and console platforms.



UI for the Anti-aliasing effect when TAA is selected

Properties

Property:	Function:
Jitter - Spread	The diameter (in texels) inside which jitter samples are spread. Smaller values result in crisper but more aliased output, whilst larger values result in more stable but blurrier output.
Blending - Stationary	The blend coefficient for stationary fragments. Controls the percentage of history sample blended into final color for fragments with minimal active motion.
Blending - Motion	The blending coefficient for moving fragments. Controls the percentage of history sample blended into the final color for fragments with significant active motion.

Property:	Function:
Sharpen	TAA can induce a slight loss of details in high frequency regions. Sharpening alleviates this issue.

Restrictions

Unsupported in VR

Requirements

Motion vectors

Depth texture

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

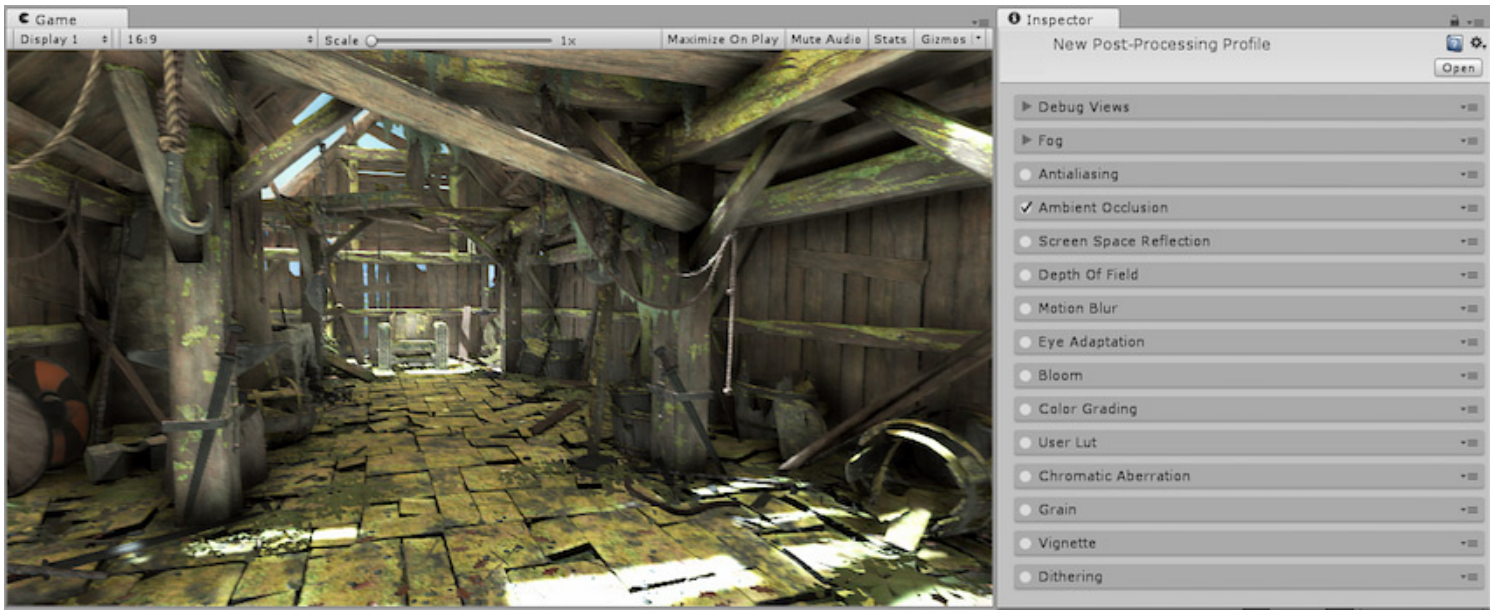
Ambient Occlusion

Leave feedback

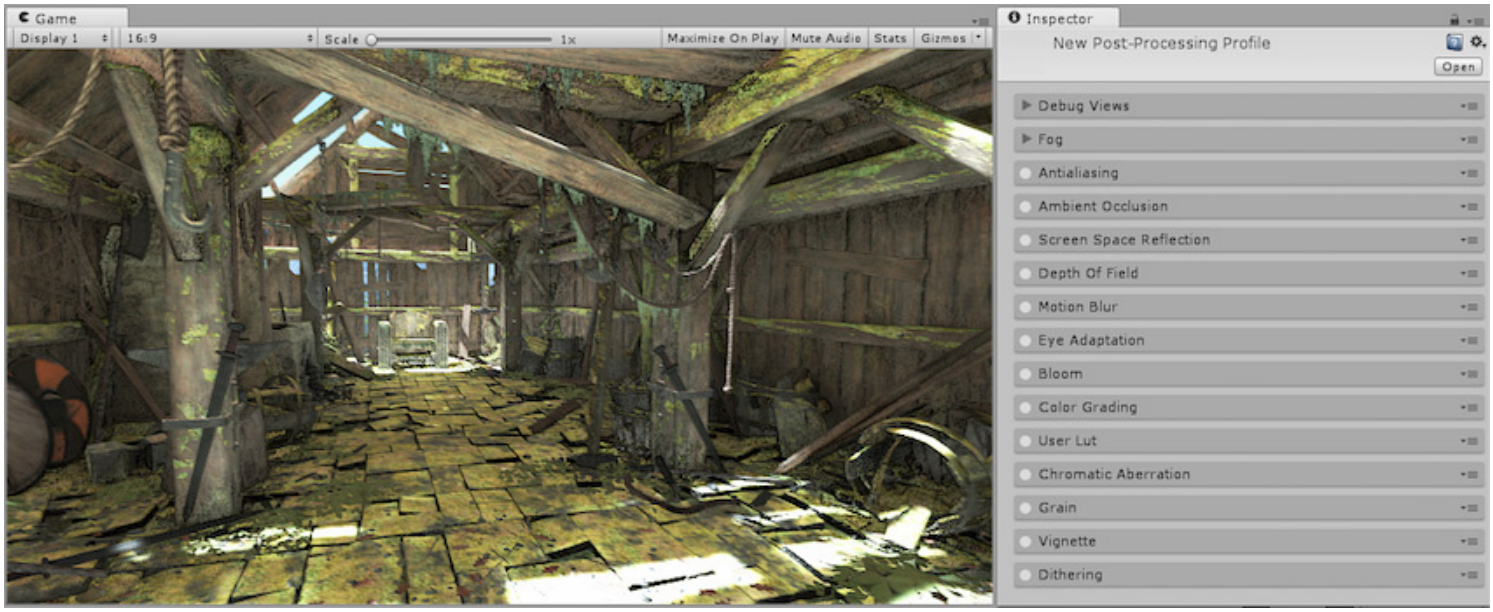
The effect descriptions on this page refer to the default effects found within the post-processing stack.

The Ambient Occlusion post-processing effect approximates Ambient Occlusion in real time as a full-screen post-processing effect. It darkens creases, holes, intersections and surfaces that are close to each other. In real life, such areas tend to block out or occlude ambient light, hence they appear darker.

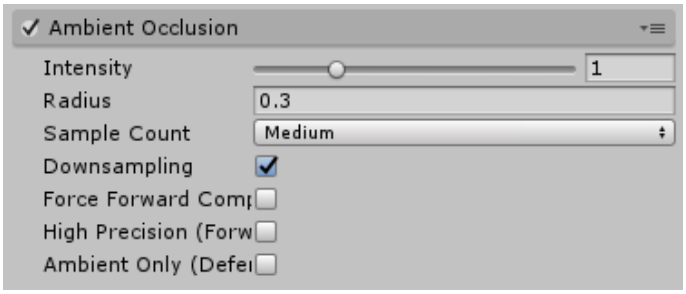
Note that the Ambient Occlusion effect is quite expensive in terms of processing time and generally should only be used on desktop or console hardware. Its cost depends purely on screen resolution and the effects parameters and does not depend on **scene** complexity as true Ambient Occlusion would.



Scene with Ambient Occlusion.



Scene without Ambient Occlusion. Note the differences at geometry intersections.



UI for Ambient Occlusion

Properties

Property:	Function:
Intensity	Degree of darkness produced by the effect.
Radius	Radius of sample points, which affects extent of darkened areas.
Sample Count	Number of sample points, which affects quality and performance.
Downsampling	Halves the resolution of the effect to increase performance at the cost of visual quality.
Force Forward Compatibility	Forces compatibility with Forward rendered objects when working with the Deferred **rendering path**.
High Precision (Forward)	Toggles the use of a higher precision depth texture with the **forward rendering** path (may impact performances). Has no effect with the deferred **rendering** path.
Ambient Only	Enables the ambient-only mode in that the effect only affects ambient lighting. This mode is only available with the Deferred rendering path and **HDR** rendering.

Optimisation

Reduce Radius size

Reduce Sample Count

Enable Downsampling

If using deferred rendering, disable Force Forward Compatibility (will cause forward rendered object to not be used when calculating Ambient Occlusion

If using forward rendering, disable High Precision (will cause the effect to use a lower precision depth texture, impacting visual quality)

Details

Beware that this effect can be quite expensive, especially when viewed very close to the **camera**. For that reason it is recommended to always enable Downsampling and favor a low radius setting. With a low radius the Ambient Occlusion effect will only sample **pixels** that are close, in clip space, to the source pixel, which is good for performance as they can be cached efficiently. With higher radiuses, the generated samples will be further away from the source pixel and won't benefit from caching thus slowing down the effect. Because of the **camera__'s perspective, objects near the front plane will use larger radiuses than those far away, so computing the Ambient Occlusion pass for an object close to the** camera__ will be slower than for an object further away that only occupies a few **pixels** on screen.

When working with the Deferred rendering path, you have the possibility to render the ambient occlusion straight to the ambient G-Buffer so that it's taken into account by Unity during the lighting pass. Note that this setting requires the camera to have HDR enabled.

When working with the Forward rendering path you may experience some quality issues in regards to depth precision. You can overcome these issues by toggling High Precision but only do it if you actually need it as it will lower performances.

Requirements

Depth & Normals texture

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

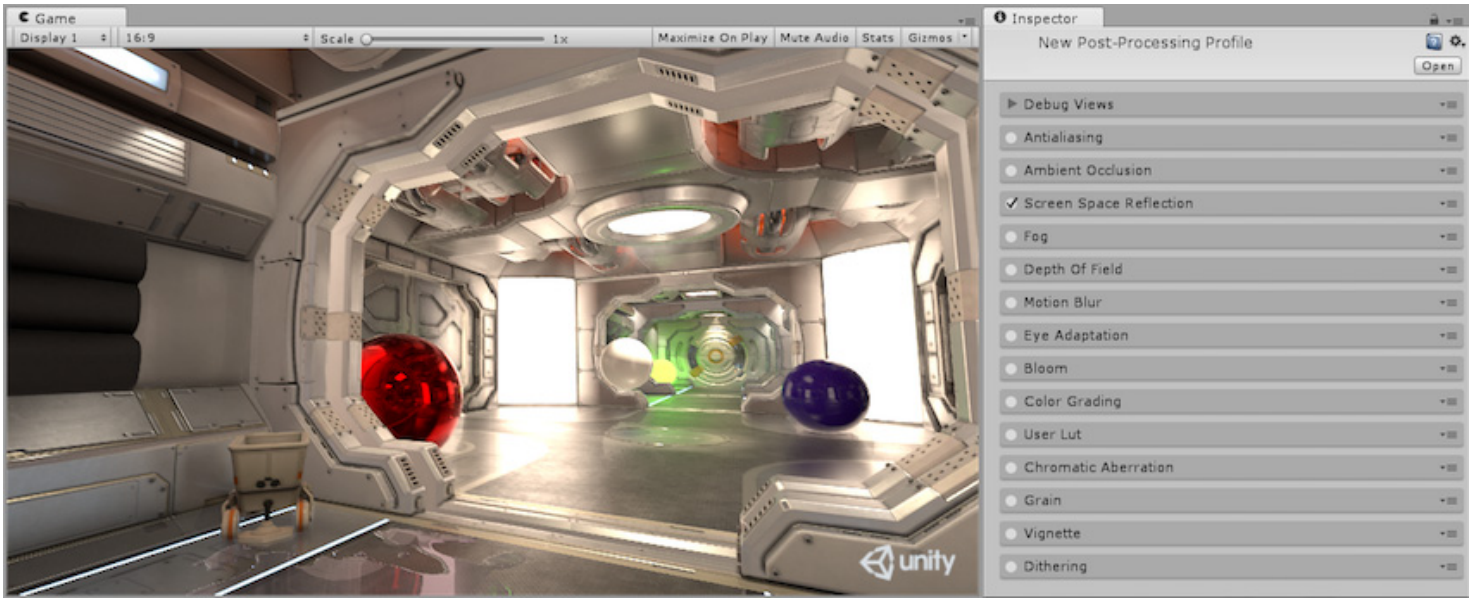
Screen Space Reflection

Leave feedback

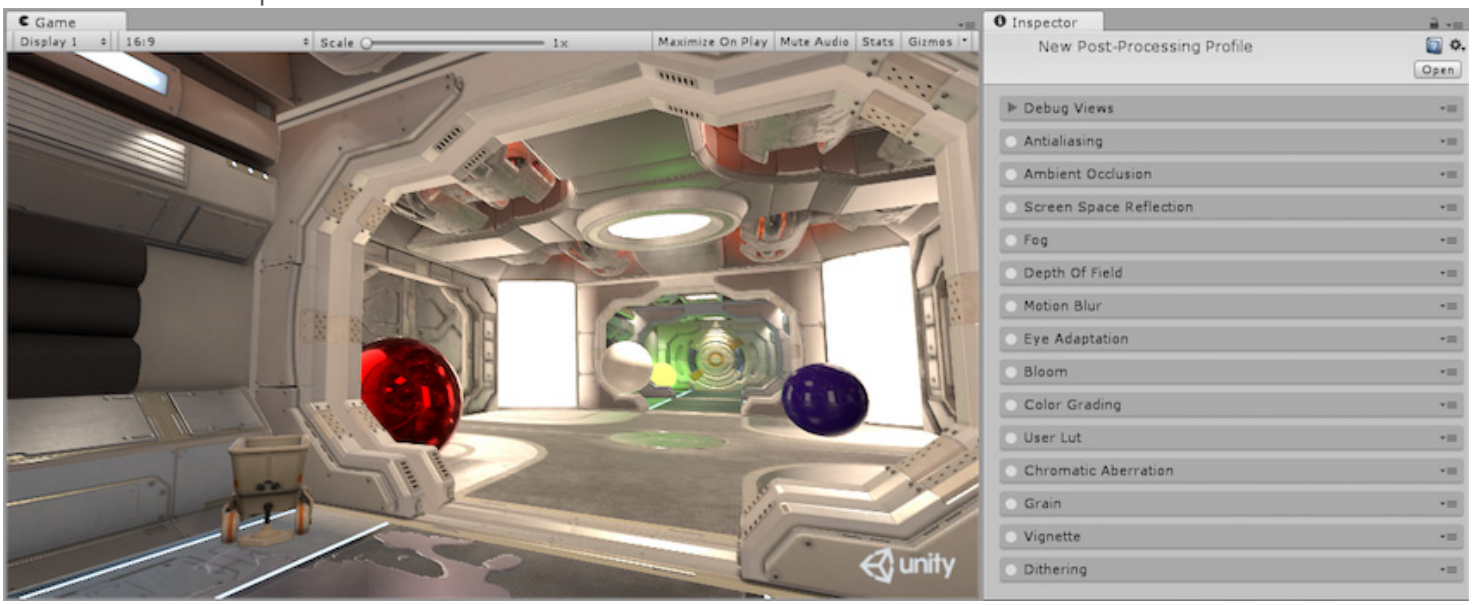
The effect descriptions on this page refer to the default effects found within the post-processing stack.

Screen Space Reflection is a technique for reusing screen space data to calculate reflections. It is commonly used to create more subtle reflections such as on wet floor surfaces or in puddles.

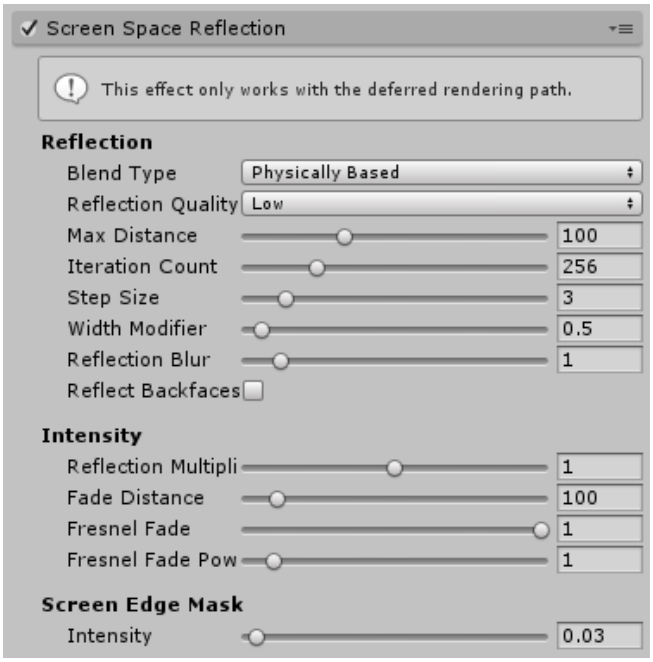
Screen Space Reflection is an expensive technique, but when used correctly can give great results. Screen Space Reflection is only available in the deferred rendering path as it relies on the Normals G-Buffer. As it is an expensive effect it is not recommended to be used on mobile.



Scene with Screen Space Reflection



Scene without reflections



UI for Screen Space Reflection

Properties

Property:	Function:
Blend Type	How the reflections are blended into the render.
Reflection Quality	The size of the buffer used for resolve. Half resolution SSR is much faster, but less accurate.
Max Distance	Maximum reflection distance in world units.
Iteration Count	Maximum raytracing length.
Step Size	Ray tracing coarse step size. Higher traces farther, lower gives better quality silhouettes.
Width Modifier	Typical thickness of columns, walls, furniture, and other objects that reflection rays might pass behind.
Reflection Blur	Blurriness of reflections.
Reflect Backfaces	Renders the **scene** by culling all front faces and uses the resulting texture for estimating what the backfaces might look like when a point on the depth map is hit from behind.
Reflection Multiplier	Nonphysical multiplier for the SSR reflections. 1.0 is physically based.
Fade Distance	How far away from the Max Distance to begin fading SSR.
Fresnel Fade	Amplify Fresnel fade out. Increase if floor reflections look good close to the surface and bad farther 'under' the floor.
Fresnel Fade Power	Higher values correspond to a faster Fresnel fade as the reflection changes from the grazing angle.
(Screen Edge Mask) Intensity	Higher values fade out SSR near the edge of the screen so that reflections don't pop under camera motion.

Optimisation

Disable Reflect Backfaces

Reduce Reflection Quality

Reduce Iteration Count (increase step size to compensate)

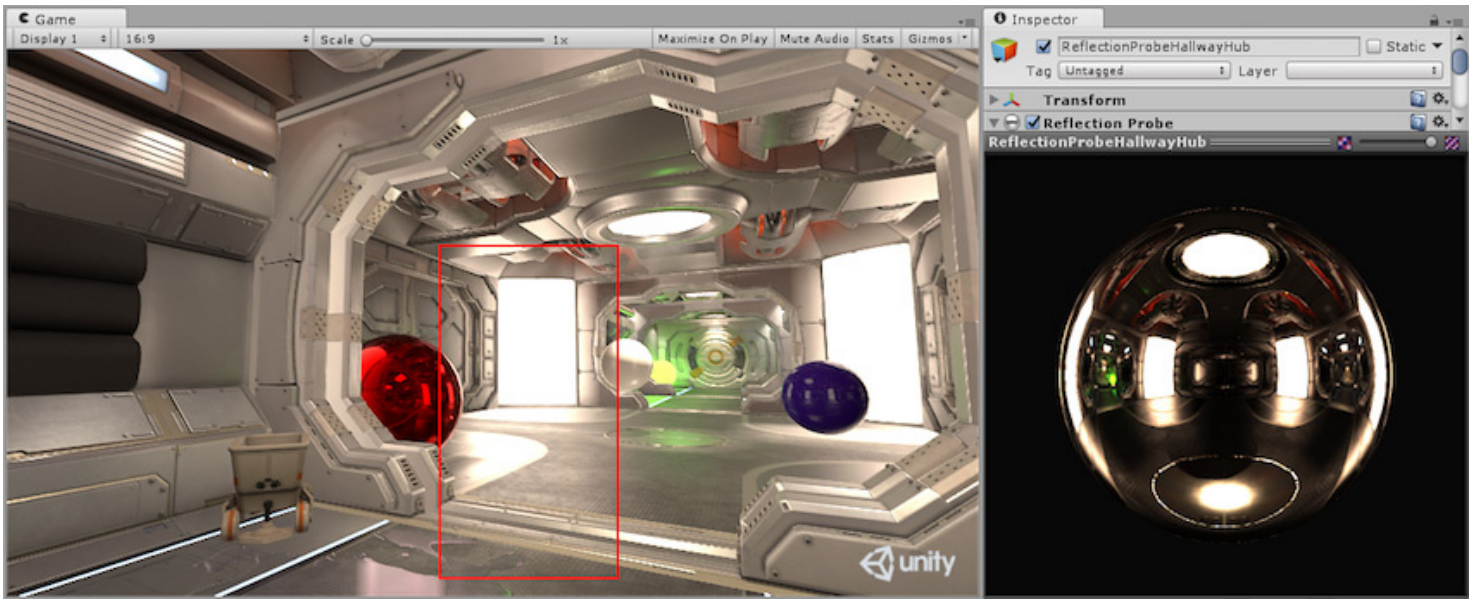
Use Additive Reflection

Restrictions

Unsupported in VR

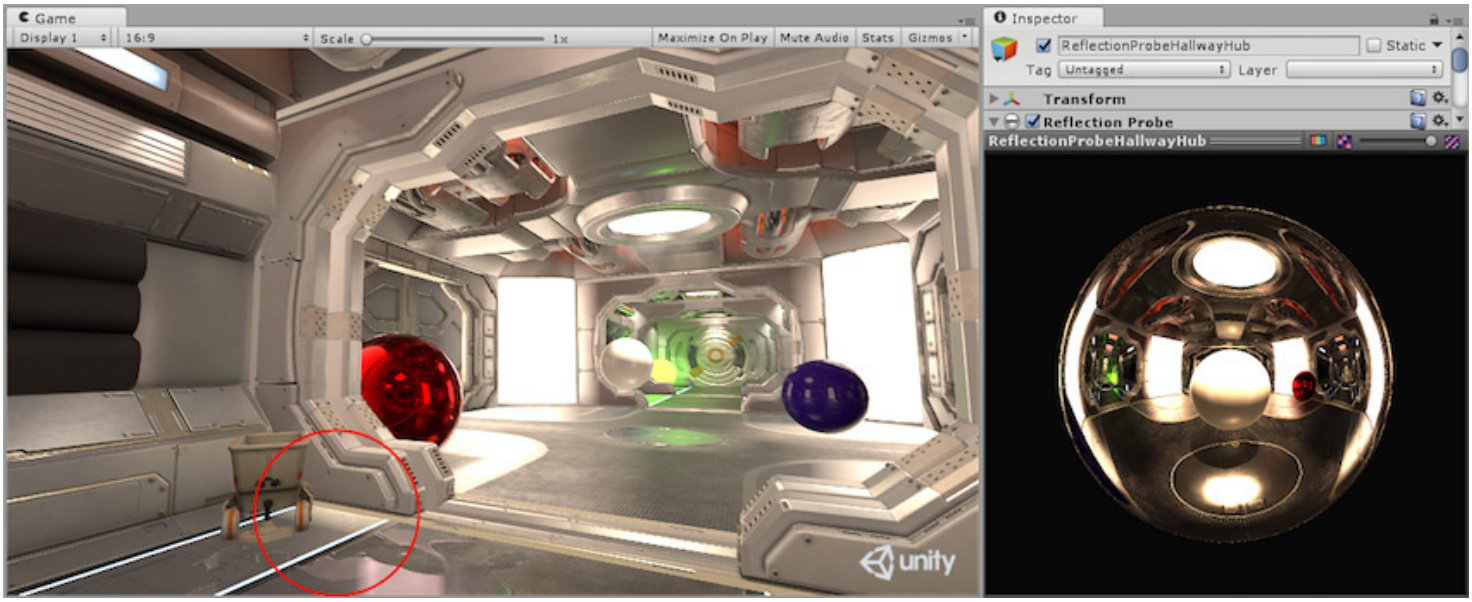
Details

Screen Space Reflection can be used to obtain more detailed reflections than other methods such as Cubemaps or Reflection Probes. Objects using Cubemaps for reflection are unable to obtain self reflection and Reflection Probe reflections are limited in their accuracy.



Scene using the baked Reflection Probes

In the above image you can see inaccurate reflection in the red-highlighted area. This is due to the translation between the Camera and Reflection Probe. Also notice that as this Reflection Probe is baked it is unable to reflect dynamic object such as the colored spheres.



With realtime Reflection Probes (pictured above) dynamic objects are captured but, like in the example above, the position of the reflection is incorrect. In the red-highlighted area you can see the reflection of the white sphere.

Comparing these to the image at the top of the page (using Screen Space Reflection) we can clearly see the disparity in reflection accuracy, however these methods are much less expensive and should always be used when such accuracy is not necessary.

Screen Space Reflection is calculated by ray-marching from reflection points on the depth map to other surfaces. A reflection vector is calculated for each reflective point in the depth buffer. This vector is marched in steps until an intersection is found with another point on the depth buffer. This second point is then draw to the original point as a reflection.

Reducing **Iteration Count** reduces the amount of amount of times the ray is tested against the depth buffer, reducing the cost substantially. However, doing so will shorten the overall depth that is tested resulting in shorter reflections. Increasing the **Step Size** increases the distance between these tests, regaining the overall depth but reducing precision.

When using the **Physically Based Blend Type** the BRDF of the reflective material is sampled and used to alter the resulting reflection, this process is expensive but results in more realistic reflections, especially for rougher surfaces.

When using **Reflect Backfaces** the effect will also raytrace in the opposite direction in attempt to approximate the reflection of the back of an object. This process vastly increases the cost of the effect but can be used to get approximate reflection on reflective objects with other objects in front of them.

Requirements

Deferred rendering path

Depth & Normals texture

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

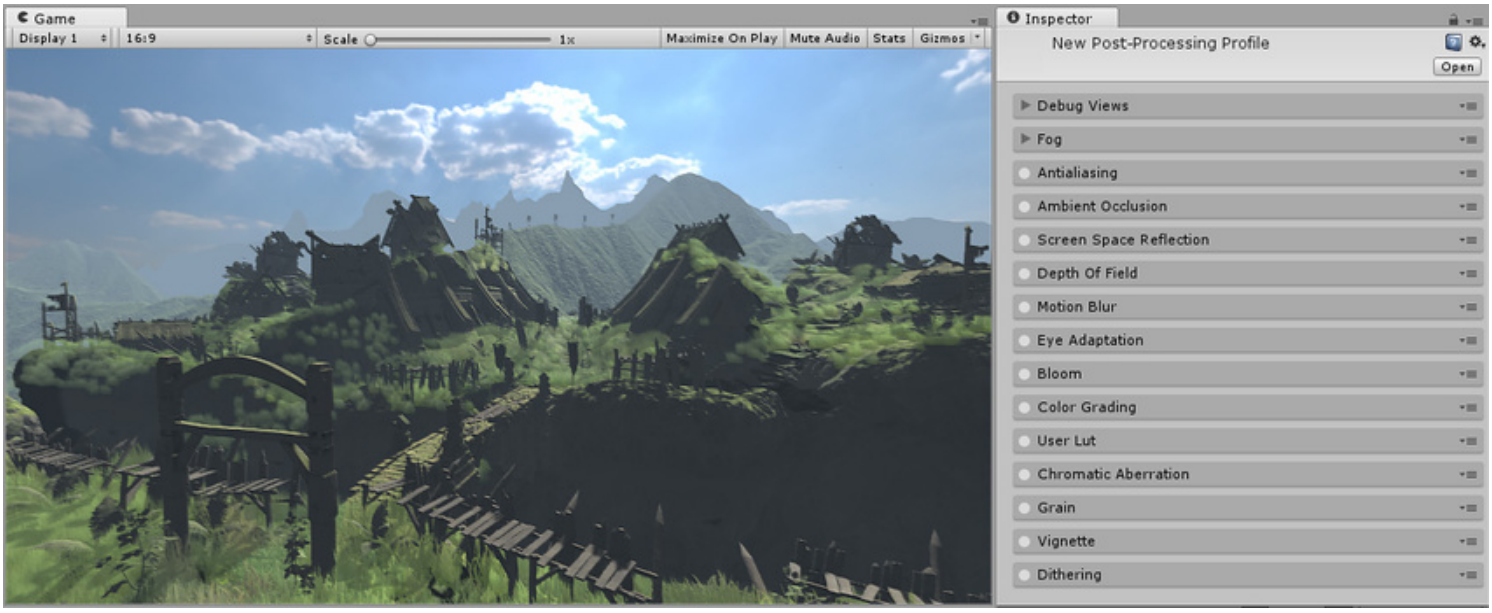
Fog

Leave feedback

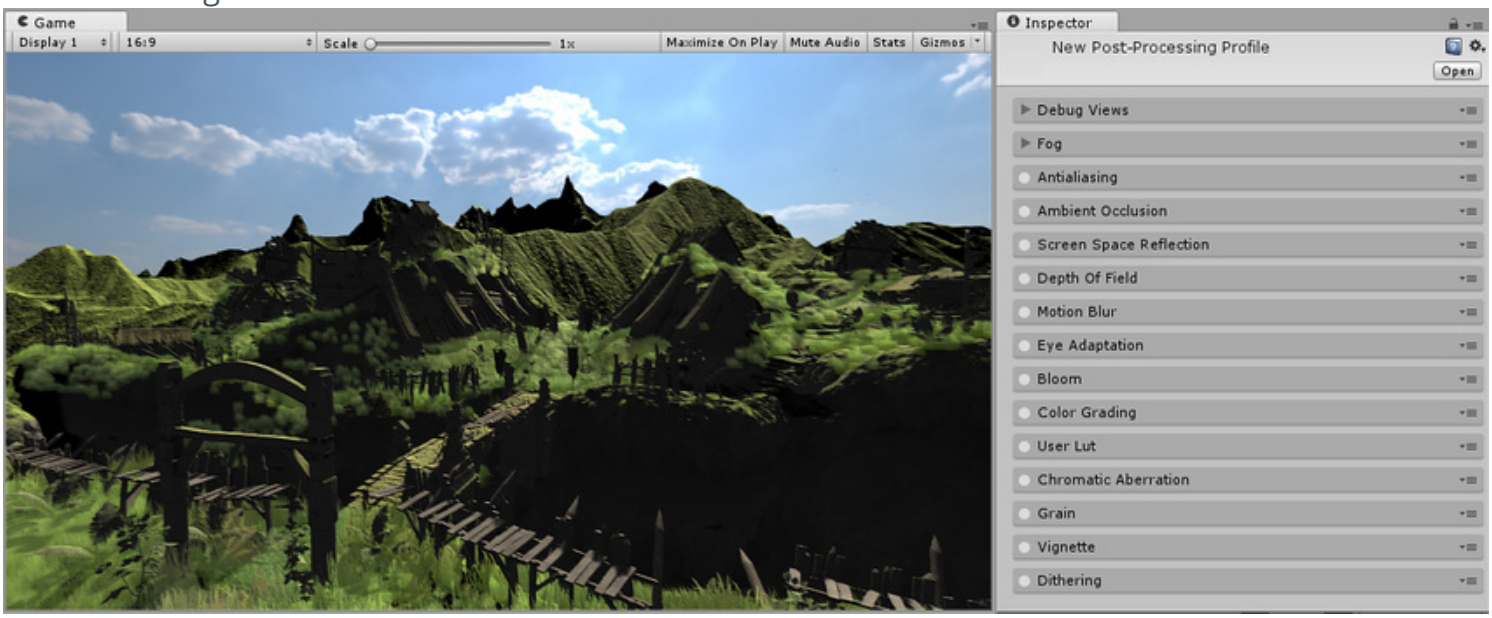
The effect descriptions on this page refer to the default effects found within the post-processing stack.

Fog is the effect of overlaying a color onto objects dependant on the distance from the **camera**. This is used to simulate fog or mist in outdoor environments and is also typically used to hide clipping of objects when a __camera__'s far clip plane has been moved forward for performance.

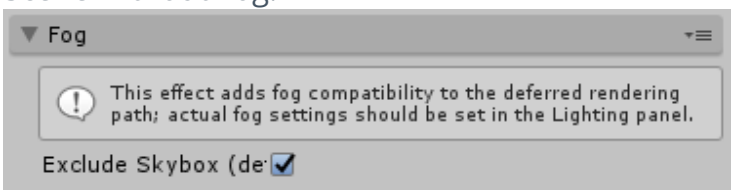
The Fog effect creates a screen-space fog based on the camera's depth texture. It supports Linear, Exponential and Exponential Squared fog types. Fog settings should be set in the **Scene** tab of the **Lighting** window.



Scene with Fog



Scene without Fog.



UI for Fog

Properties

Property:	Function:
Exclude Skybox	Should the fog affect the **skybox**?

Details

This effect is only applied in the deferred **rendering path**. When using either **rendering path** fog should be applied to forward rendered objects using the Fog found in **Scene** Settings. The parameters for the Post-processing fog are mirrored from the Fog parameters set in the **Scene** tab of the **Lighting** window. This ensures forward rendered objects will always receive the same fog when **rendering** in deferred.

Requirements

Depth texture

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

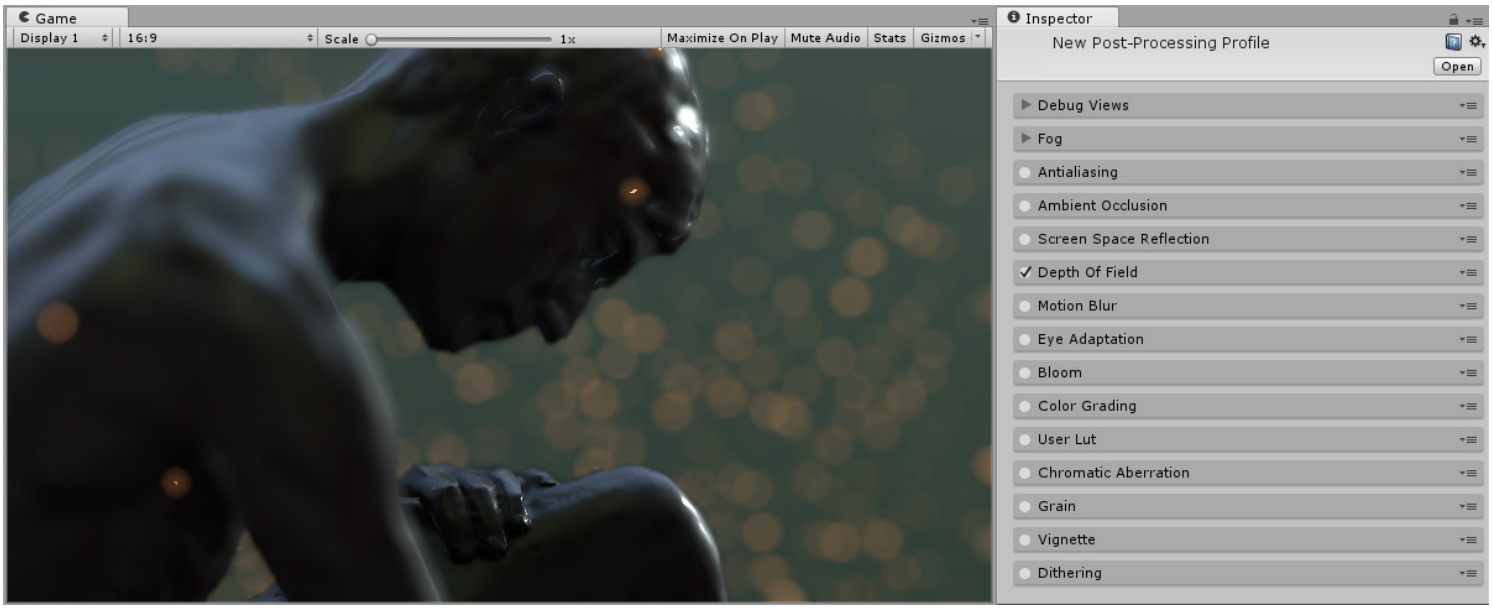
Depth of Field

Leave feedback

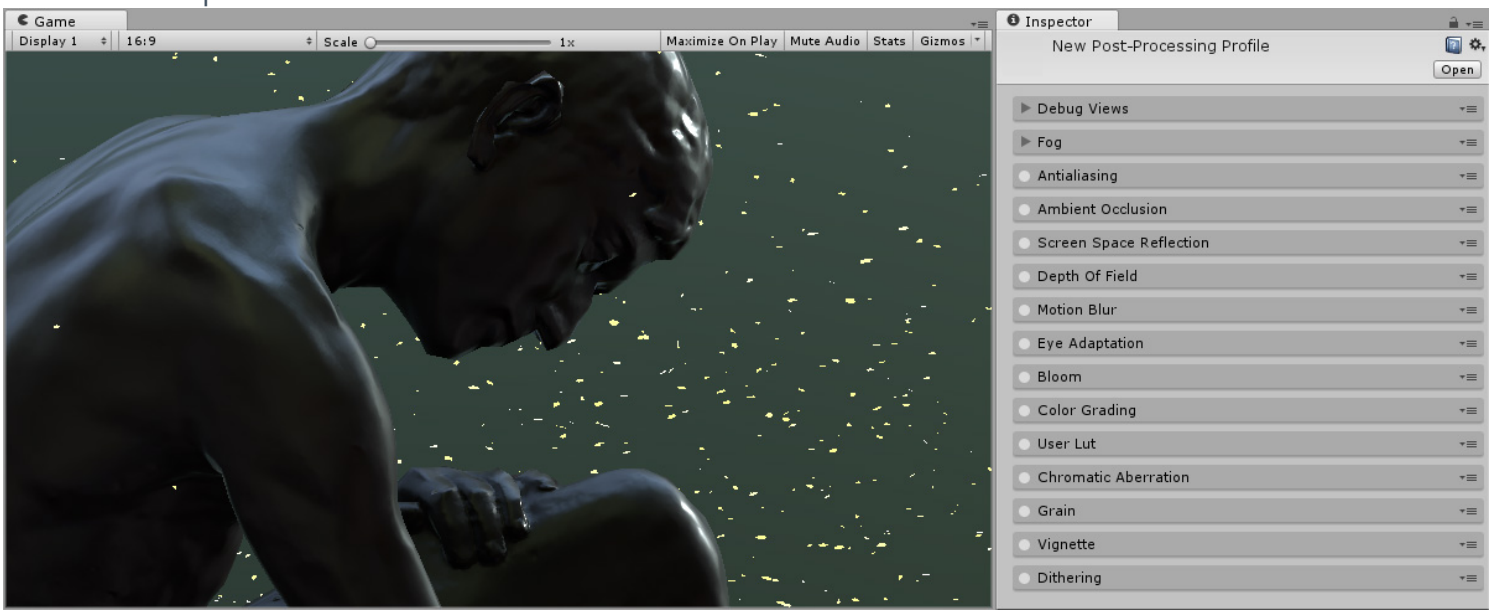
The effect descriptions on this page refer to the default effects found within the post-processing stack.

Depth of Field is a common post-processing effect that simulates the focus properties of a **camera** lens. In real life, a **camera** can only focus sharply on an object at a specific distance; objects nearer or farther from the **camera** will be somewhat out of focus. The blurring not only gives a visual cue about an object's distance but also introduces Bokeh which is the term for pleasing visual artifacts that appear around bright areas of the image as they fall out of focus.

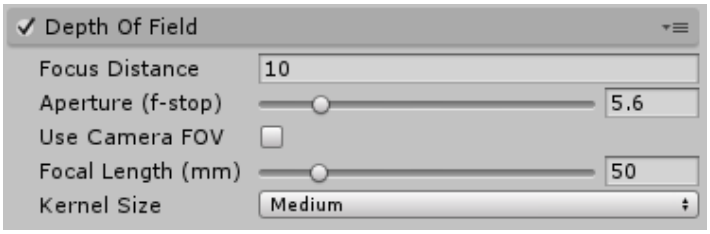
An example of **Depth of Field** effect can be seen in the following images, displaying the results of a focused midground but a defocused background and foreground.



Scene with Depth of Field.



Scene without Depth of Field.



UI for Depth of Field

Properties

Property:	Function:
Focus Distance	Distance to the point of focus.
Aperture	Ratio of the aperture (known as f-stop or f-number). The smaller the value is, the shallower the depth of field is.
Focal Length	Distance between the lens and the film. The larger the value is, the shallower the depth of field is.
Use Camera FOV	Calculate the focal length automatically from the field-of-view value set on the camera.
Kernel Size	Convolution kernel size of the bokeh filter, which determines the maximum radius of bokeh. It also affects the performance (the larger the kernel is, the longer the GPU time is required).

Optimisation

Reduce Kernel Size

Requirements

Depth texture

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

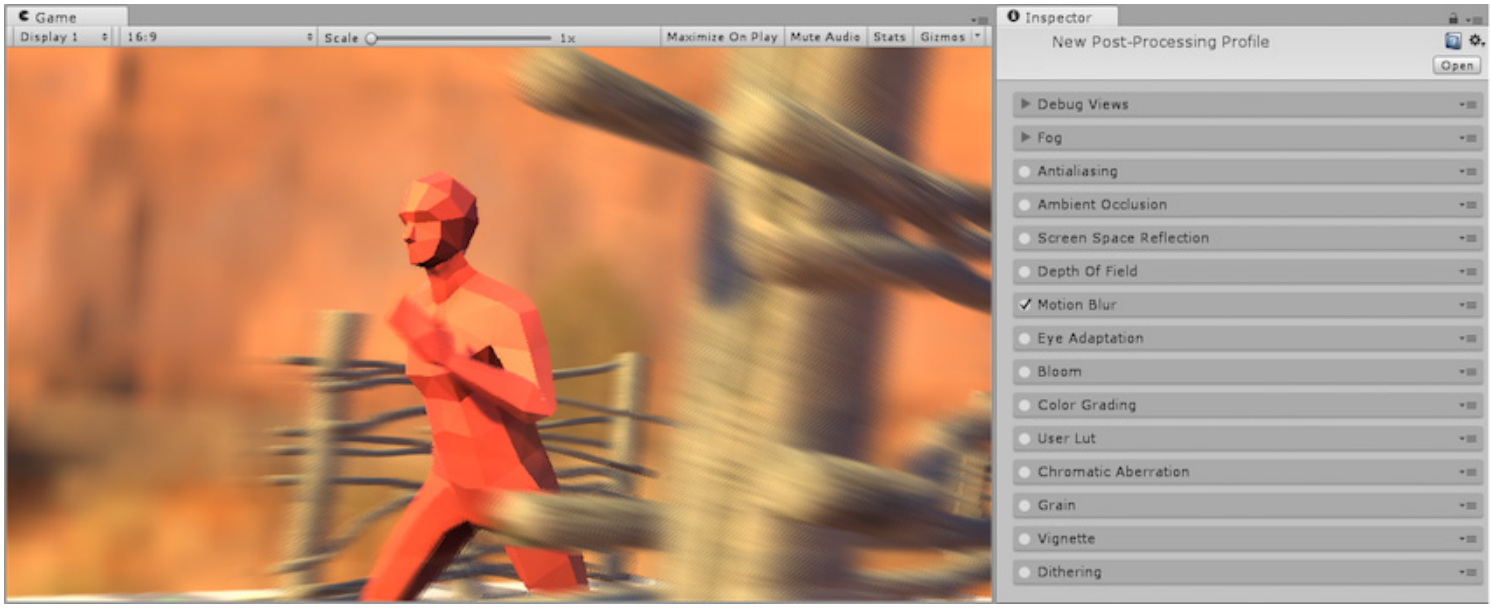
New feature in 5.6

Motion Blur

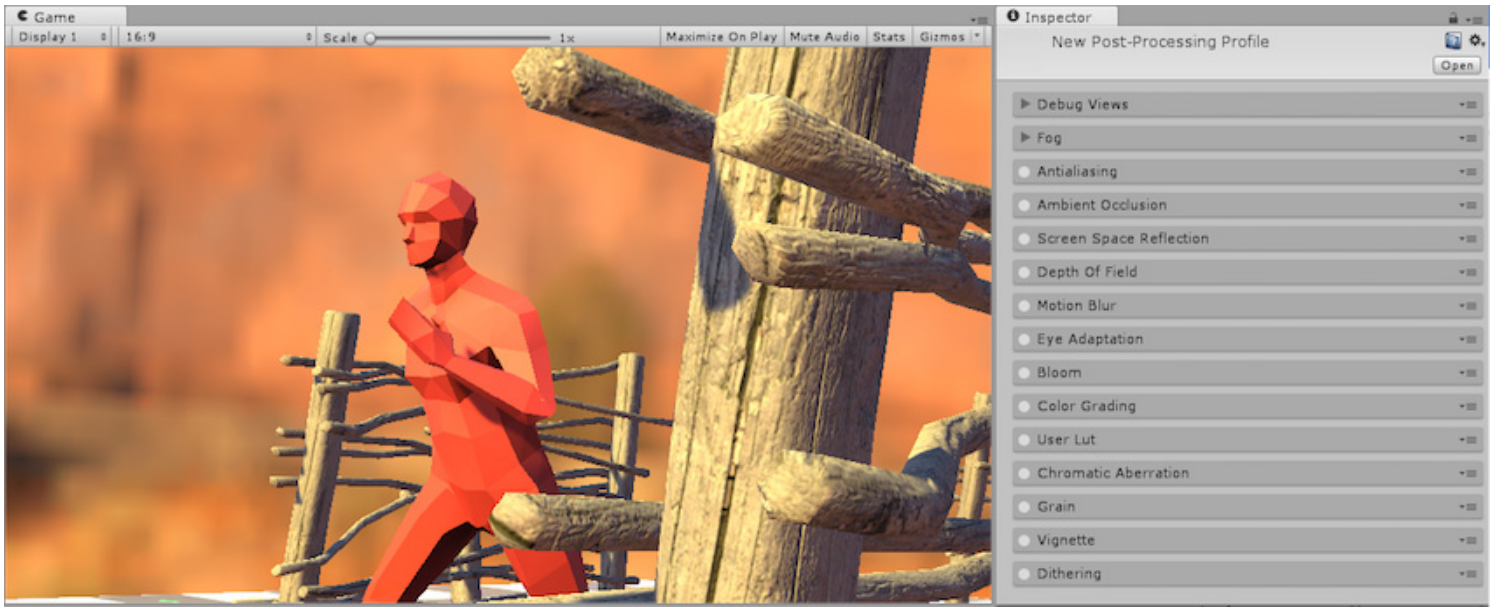
Leave feedback

The effect descriptions on this page refer to the default effects found within the post-processing stack.

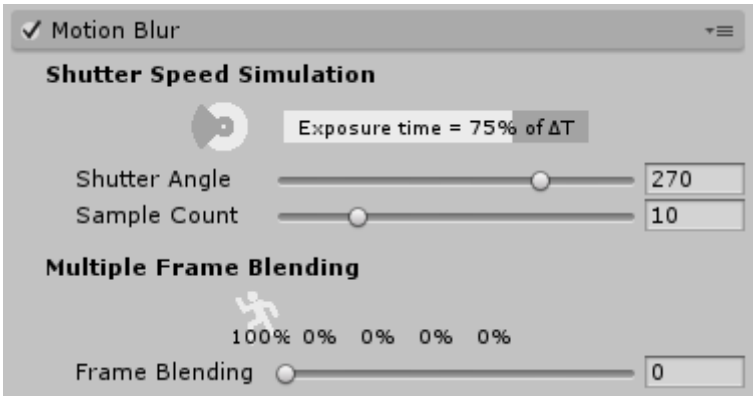
Motion Blur is a common post-processing effect that simulates the blurring of an image when objects filmed by a **camera** are moving faster than the **camera__'s exposure time. This can be caused by rapidly moving objects or a long exposure time.** Motion Blur__ is used to subtle effect in most types of games but exaggerated in some genres, such as racing games.



Scene with Motion Blur.



Scene without Motion Blur.



UI for Motion Blur

The Motion Blur techniques supplied in the post-processing stack are:

Shutter Speed Simulation

Multiple Frame Blending

Shutter Speed Simulation

Shutter Speed Simulation provides a more accurate representation of a camera's blur properties. However, as it requires Motion Vector support it is more expensive and not supported on some platforms. It is the recommended technique for desktop and console platforms. This effect approximates Motion Blur by storing the motion of **pixels** on screen in a Velocity buffer. This buffer is then used to blur **pixels** based on the distance they have moved since the last frame was drawn.

Properties

Property:	Function:
Shutter Angle	The angle of the rotary shutter. Larger values give longer exposure therefore a stronger blur effect.
Sample Count	The amount of sample points, which affects quality and performances.

Optimisation

Reduce Sample Count

Restrictions

Unsupported in VR

Requirements

Motion Vectors

Depth texture

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

Multiple Frame Blending

The **Multiple Frame Blending** effect simply multiplies the previous four frames over the current frame, weighted towards the more recent frames. Whilst this effect will work on all platforms, as it does not require Motion Vector or Depth texture support, it requires storing two history buffers (luma and chroma) of the last four frames which uses memory.

Properties

Property:	Function:
Frame Blending	The strength of multiple frame blending. The opacity of the preceding frames are determined from this coefficient and time differences.

Optimisation

N/A

Requirements

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

Eye Adaptation

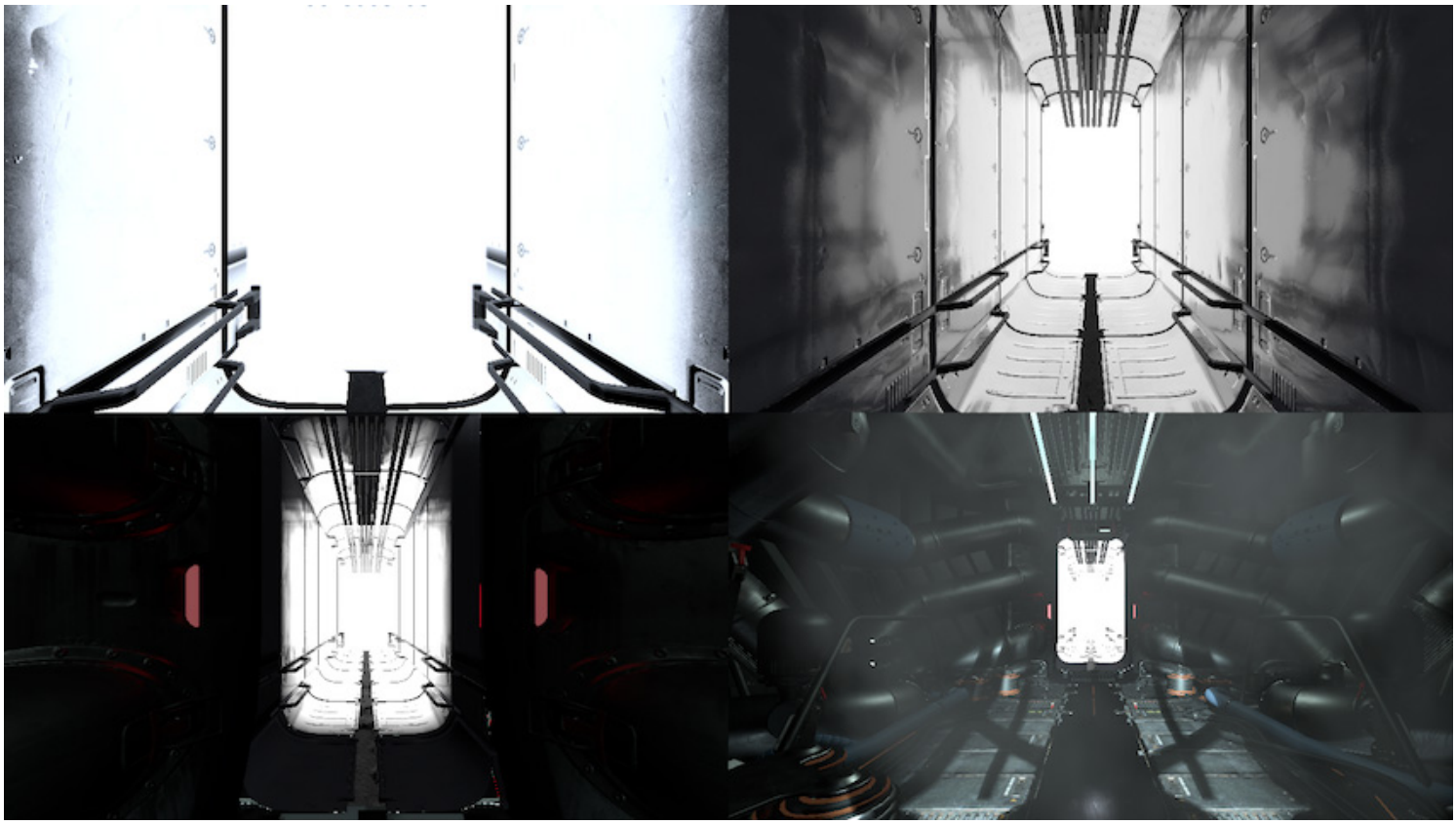
Leave feedback

The effect descriptions on this page refer to the default effects found within the post-processing stack.

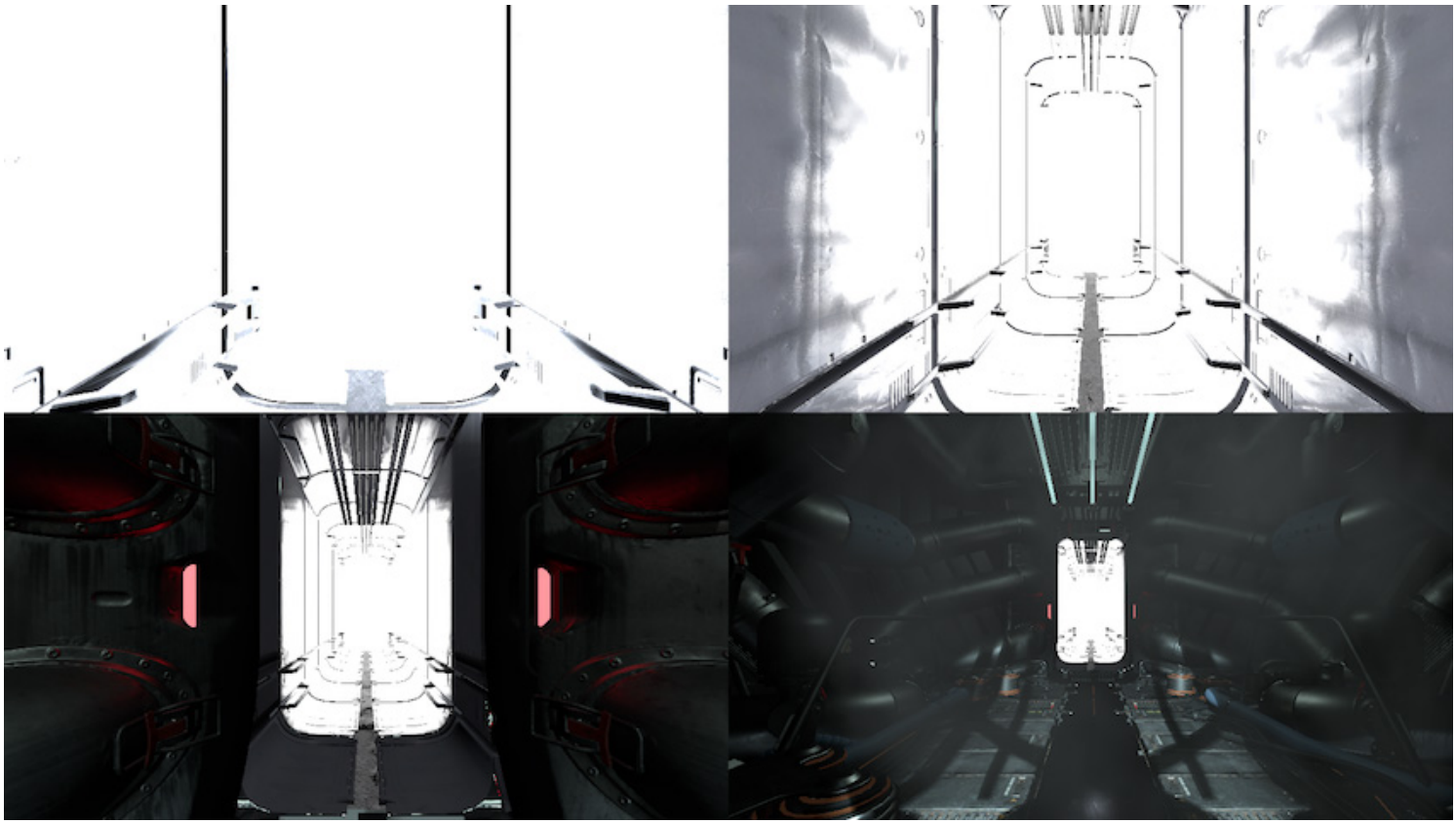
In ocular physiology, adaptation is the ability of the eye to adjust to various levels of darkness and light. The human eye can function from very dark to very bright levels of light. However, in any given moment of time, the eye can only sense a contrast ratio of roughly one millionth of the total range. What enables the wider reach is that the eye adapts its definition of what is black.

This effect dynamically adjusts the exposure of the image according to the range of brightness levels it contains. The adjustment takes place gradually over a period of time, so the player can be briefly dazzled by bright outdoor light when, say, emerging from a dark tunnel. Equally, when moving from a bright **scene** to a dark one, the “eye” takes some time to adjust.

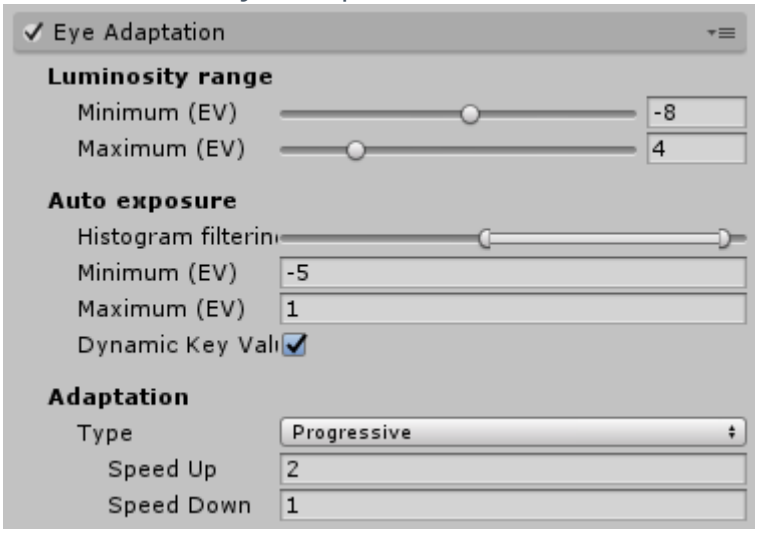
Internally, this effect generates a histogram on every frame and filters it to find the average luminance value. This histogram, and as such the effect, requires Compute shader support.



Scene with Eye Adaptation.



Scene without Eye Adaptation.



UI for Eye Adaptation

Properties

Property:	Function:
Luminosity Range	
Minimum (EV)	Lower bound for the brightness range of the generated histogram (in EV). The bigger the spread between min & max, the lower the precision will be.
Maximum (EV)	Upper bound for the brightness range of the generated histogram (in EV). The bigger the spread between min & max, the lower the precision will be.
Auto exposure	

Property:	Function:
Histogram Filtering	These values are the lower and upper percentages of the histogram that will be used to find a stable average luminance. Values outside of this range will be discarded and wont contribute to the average luminance.
Minimum (EV)	Minimum average luminance to consider for auto exposure (in EV).
Maximum (EV)	Maximum average luminance to consider for auto exposure (in EV).
Dynamic Key Value	Set this to true to let Unity handle the key value automatically based on average luminance.
Key Value	Exposure bias. Use this to offset the global exposure of the scene.
Adaptation	
Adaptation Type	Use Progressive if you want the auto exposure to be animated. Use Fixed otherwise.
Speed Up	Adaptation speed from a dark to a light environment.
Speed Down	Adaptation speed from a light to a dark environment.

Details

The **Luminosity Range Minimum/Maximum** values are used to set the available histogram range in EV units. The larger the range is, the less precise it will be. The default values should work fine for most cases, but if you're working with a very dark scene you'll probably want to drop both values to focus on darker areas.

Use the **Histogram Filtering** range to exclude the darkest and brightest part of the image. To compute an average luminance you generally don't want very dark and very bright **pixels** to contribute too much to the result. Values are in percent.

Auto Exposure Minimum/Maximum values clamp the computed average luminance into a given range.

Tweak Exposure Compensation (also known as **Key Value**) to adjust the luminance offset.

You can also set the **Adaptation Type** to **Fixed** if you don't need the eye adaptation effect and it will behave like an auto-exposure setting.

It is recommended to use the Eye Adaptation Debug view when setting up this effect.

Requirements

Compute shader

Shader model 5

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with limited editorial review

New feature in 5.6

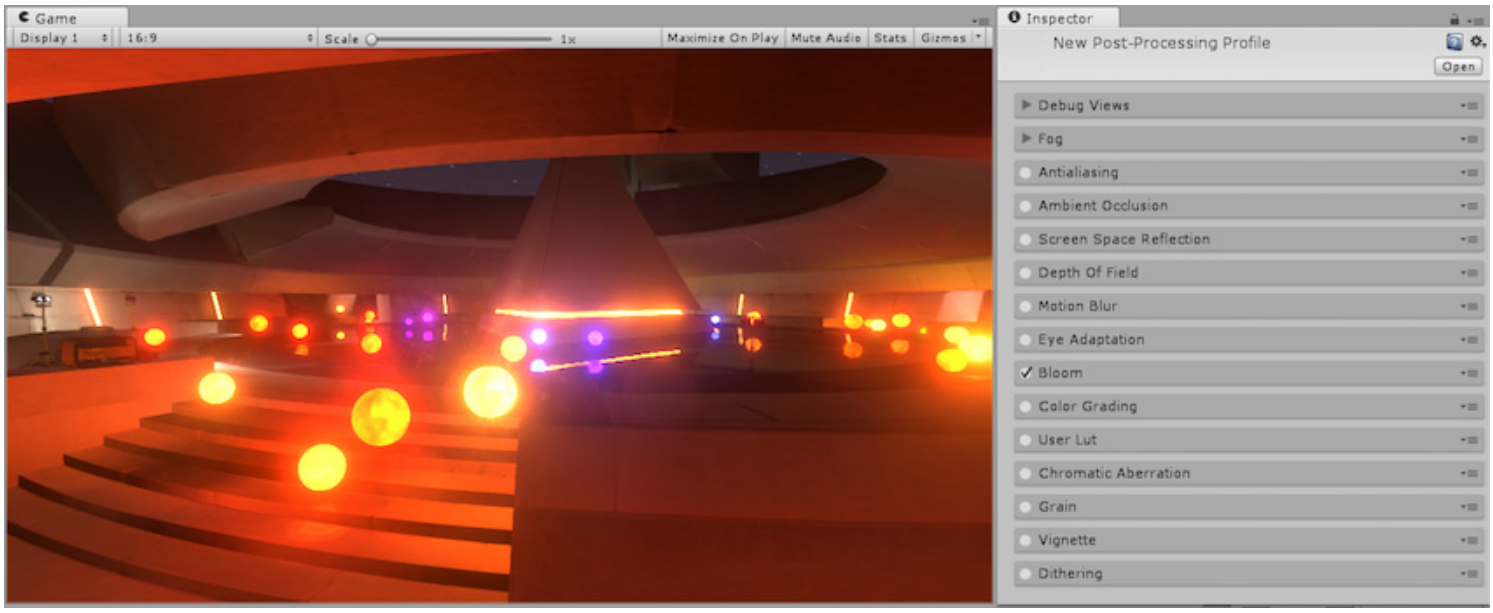
Bloom

Leave feedback

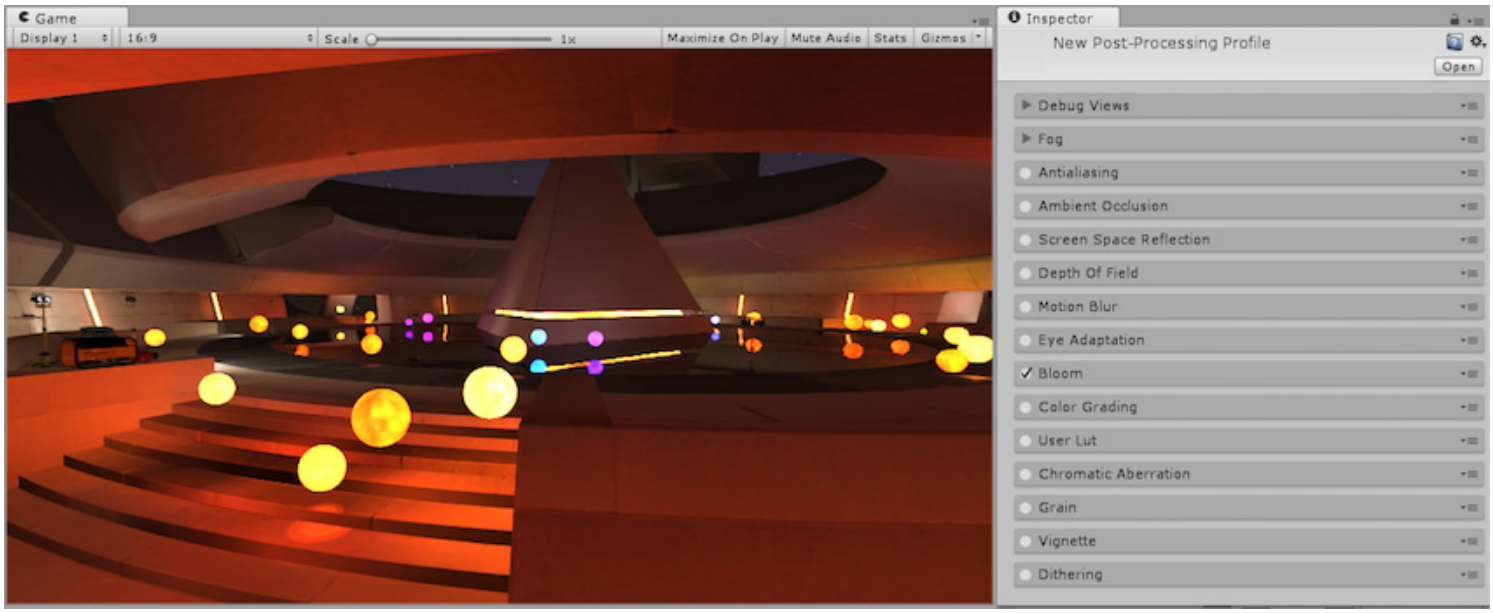
The effect descriptions on this page refer to the default effects found within the post-processing stack.

Bloom is an effect used to reproduce an imaging artifact of real-world cameras. The effect produces fringes of light extending from the borders of bright areas in an image, contributing to the illusion of an extremely bright light overwhelming the **camera** or eye capturing the **scene**.

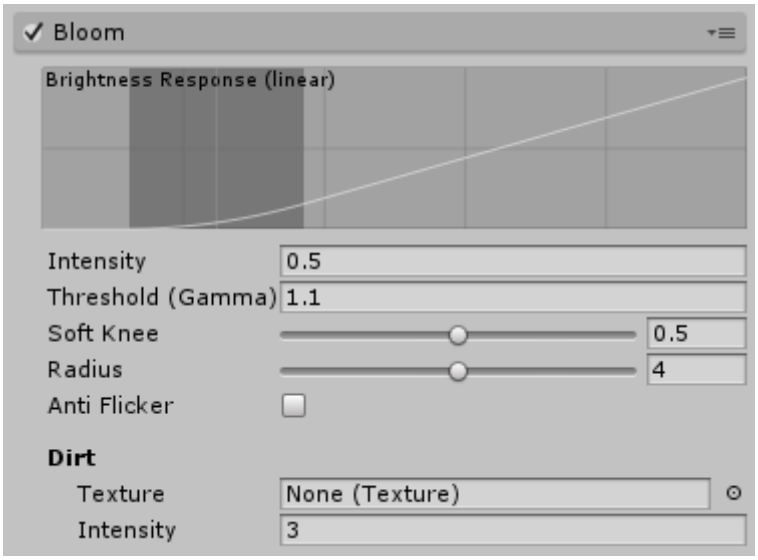
In **HDR rendering** a Bloom effect should only affects areas of brightness above LDR range (above 1) by setting the **Threshold** parameter just above this value.



Scene with Bloom.



Scene without Bloom.



UI for Bloom

Properties

Property:	Function:
Intensity	Strength of the Bloom filter.
Threshold	Filters out **pixels** under this level of brightness.
Soft Knee	Makes transition between under/over-threshold gradual (0 = hard threshold, 1 = soft threshold).
Radius	Changes extent of veiling effects in a screen resolution-independent fashion.
Anti Flicker	Reduces flashing noise with an additional filter.

Optimisation

Reduce Radius

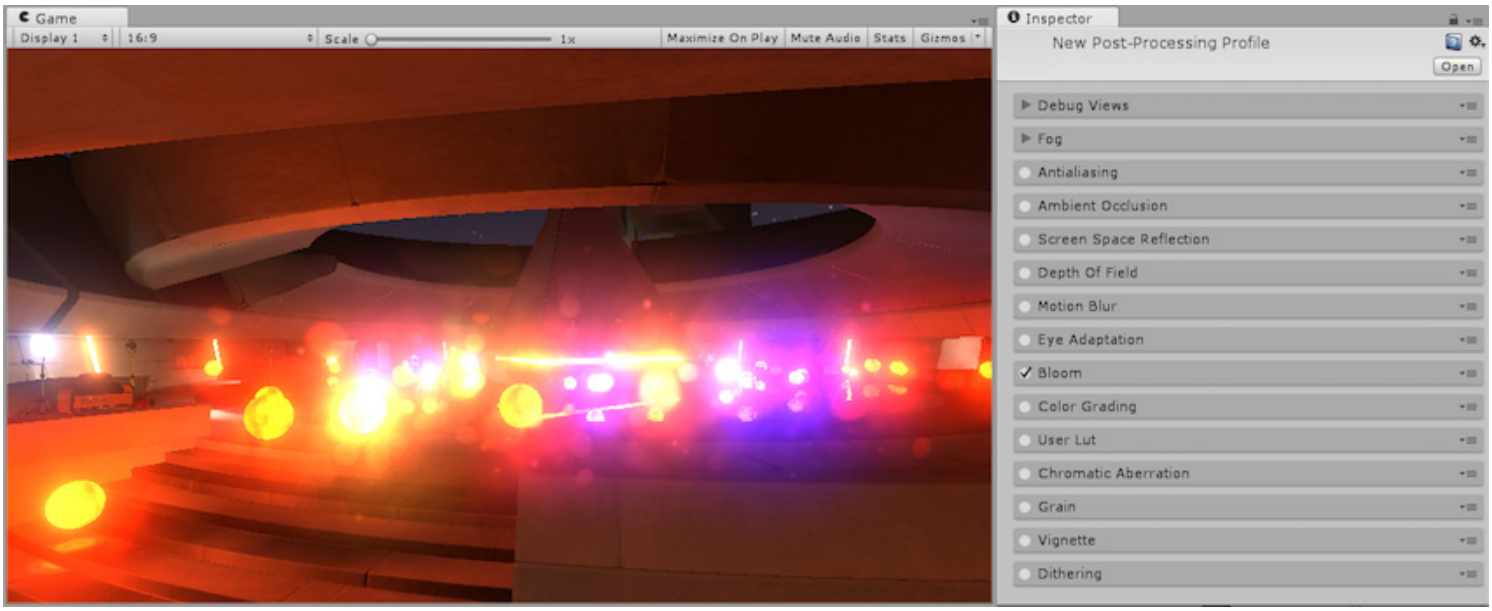
Details

With properly exposed HDR scenes, **Threshold** should be set to ~1 so that only pixels with values above 1 leak into surrounding objects. You'll probably want to drop this value when working in LDR or the effect won't be visible.

Anti Flicker reduces flashing noise, commonly known as "fireflies", by running an additional filter on the picture beforehand. This will affect performances and should be disabled when Temporal Anti-aliasing is enabled.

Lens Dirt

Lens Dirt applies a fullscreen layer of smudges or dust to diffract the Bloom effect. This is commonly used in modern **first person shooters**.



The same scene with **Lens Dirt** applied

Properties

Property:	Function:
Texture	Dirtiness texture to add smudges or dust to the lens.
Intensity	Amount of lens dirtiness

Optimisation

Reduce resolution of **Lens Dirt** texture

Details

Lens Dirt requires an input texture to use as a fullscreen layer. There are four **Lens Dirt** texture supplied in the Post-processing stack that should cover common use cases. These textures are supplied at 3840x2160 resolution for maximum quality and should be scaled dependent on project and platform. You can create custom **Lens Dirt** textures in any image editing software.

Requirements

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

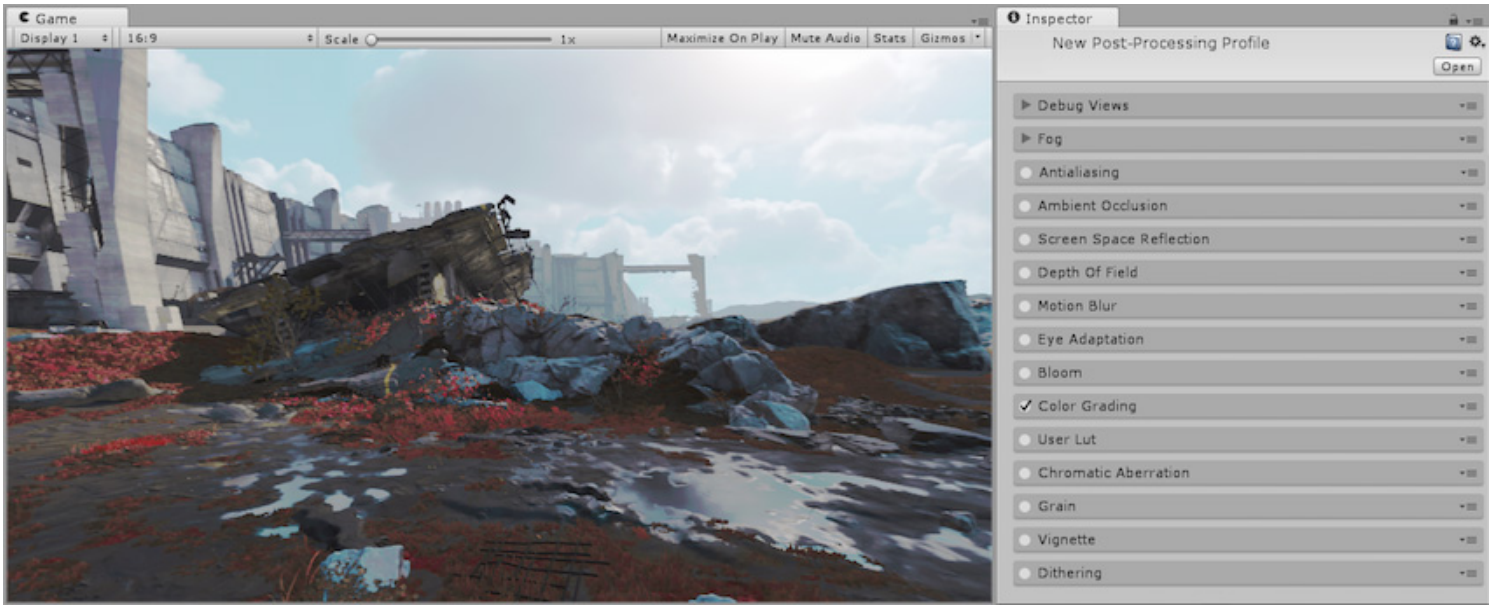
Color Grading

Leave feedback

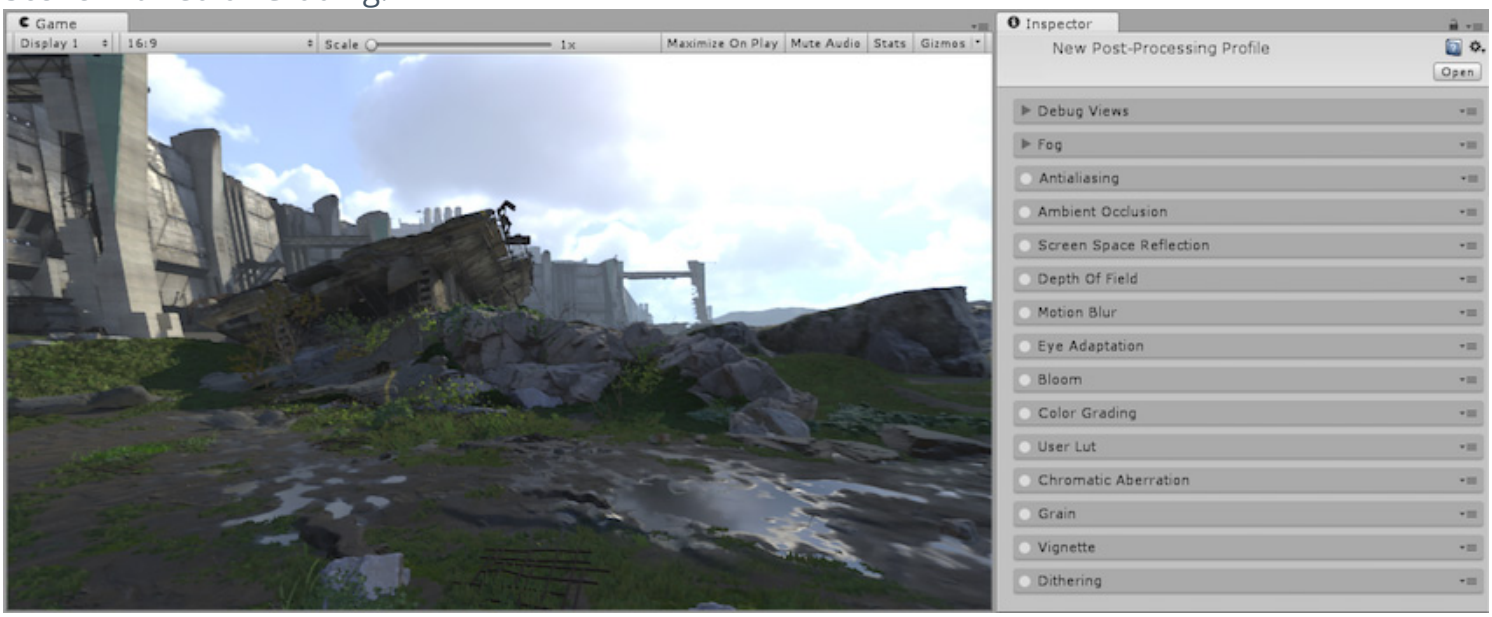
The effect descriptions on this page refer to the default effects found within the post-processing stack.

Color Grading is the process of altering or correcting the color and luminance of the final image. You can think of it like applying filters in software like Instagram.

The Color Grading tools included in the post-processing stack are fully real-time **HDR** tools and internal processing is done in the ACES color-spaces.



Scene with Color Grading.



Scene without Color Grading.

The Color Grading tools supplied in the post-processing stack come in five sections:

Tonemapping

Basic

Channel Mixer

Trackballs

Grading Curves

Requirements

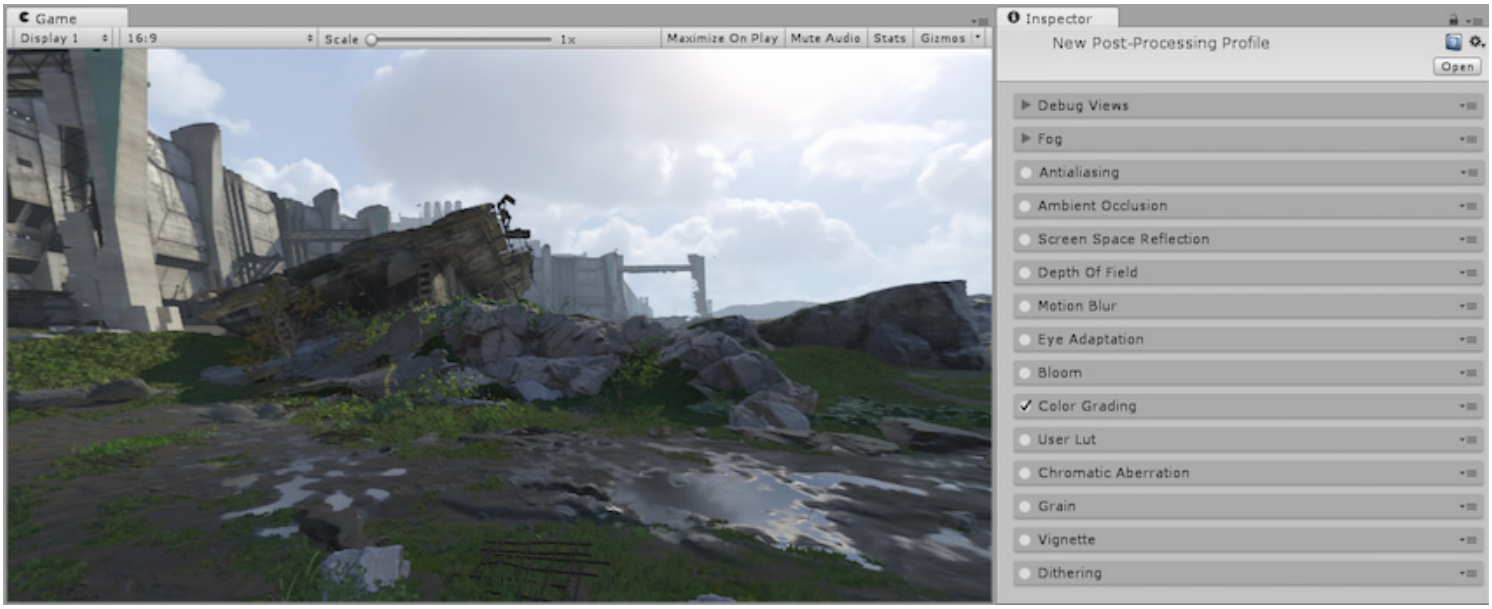
RGBAHalf Texture Format

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

Tonemapping

Tonemapping is the process of remapping HDR values of an image into a range suitable to be displayed on screen. **Tonemapping** should always be applied when using an HDR **camera**, otherwise values color intensities above 1 will be clamped at 1, altering the **scenes** luminance balance.



The same scene with Neutral Tonemapper applied (notice how the sky is not blown out).

There are three tonemapping modes supplied with the post-processing stack:

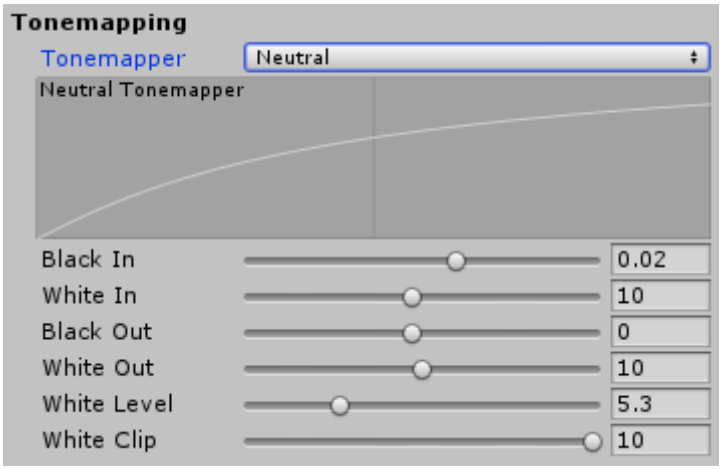
None (apply no tonemapping, select this when working in LDR)

Neutral

Filmic (ACES)

Neutral Tonemapper

The Neutral tonemapper only does range-remapping with minimal impact on color hue & saturation and is generally a great starting point for extensive color grading. Its operator is based on work by John Hable and Jim Hejl. It offers full parametric control over the tonemapping curve and is the recommended tonemapper to use in most cases.



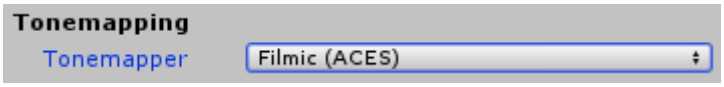
UI for Tonemapping when Neutral tonemapper is selected

Properties

Property:	Function:
Black In	Inner control point for the black point.
White In	Inner control point for the white point.
Black Out	Outer control point for the black point.
White Out	Outer control point for the white point.
White Level	Pre-curve white point adjustment.
White Clip	Post-curve white point adjustment.

Filmic (ACES) Tonemapper

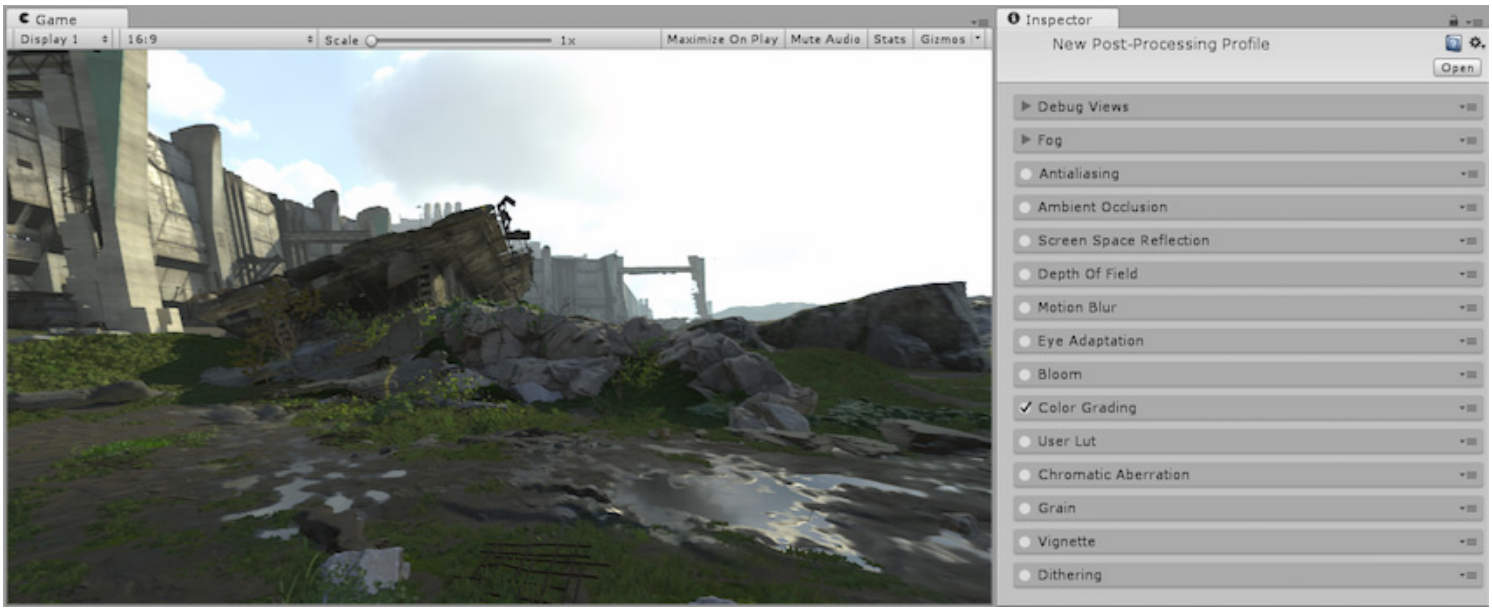
The Filmic (ACES) tonemapper uses a close approximation of the reference ACES tonemapper for a more filmic look. Because of that, it is more contrasted than Neutral and has an effect on actual color hue & saturation. This tonemapper is the simplest to use as it requires no user input to give a standard filmic look to your scene.



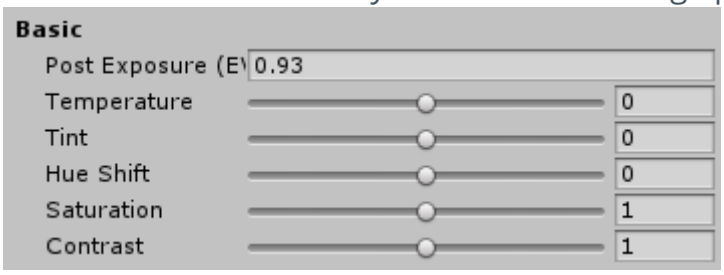
UI for Tonemapping when Filmic (ACES) tonemapper is selected

Basic

The basic section provides the simplest color grading tools such as Temperature and Contrast. This is the recommended starting point for color correction.



The same scene with only Basic Color Grading applied



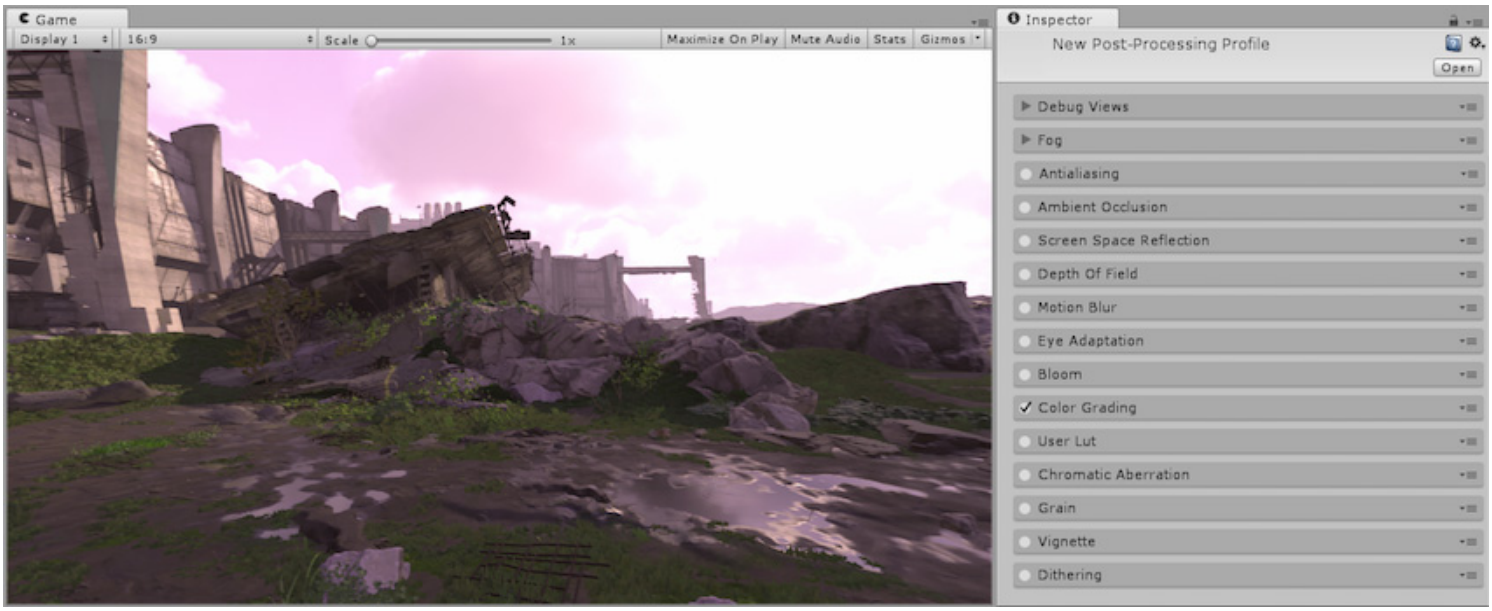
UI for Basic Color Grading

Properties

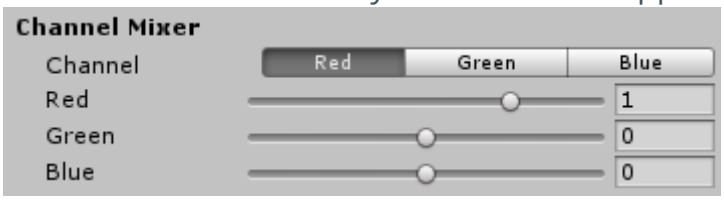
Property:	Function:
Post Exposure	Adjusts the overall exposure of the scene in EV units. This is applied after HDR effect and right before tonemapping so it won't affect previous effects in the chain.
Temperature	Sets the white balance to a custom color temperature.
Tint	Sets the white balance to compensate for a green or magenta tint.
Hue Shift	Shift the hue of all colors.
Saturation	Pushes the intensity of all colors.
Contrast	Expands or shrinks the overall range of tonal values.

Channel Mixer

The Channel Mixer is used to modify the influence of each input color channel on the overall mix of the output channel. For example, increasing the influence of the green channel on the overall mix of the red channel will adjust all areas of the image containing green (including neutral/monochrome) to become more reddish in hue.



The same scene with only Channel Mixer applied (increased blue influence on red).



UI for Channel Mixer

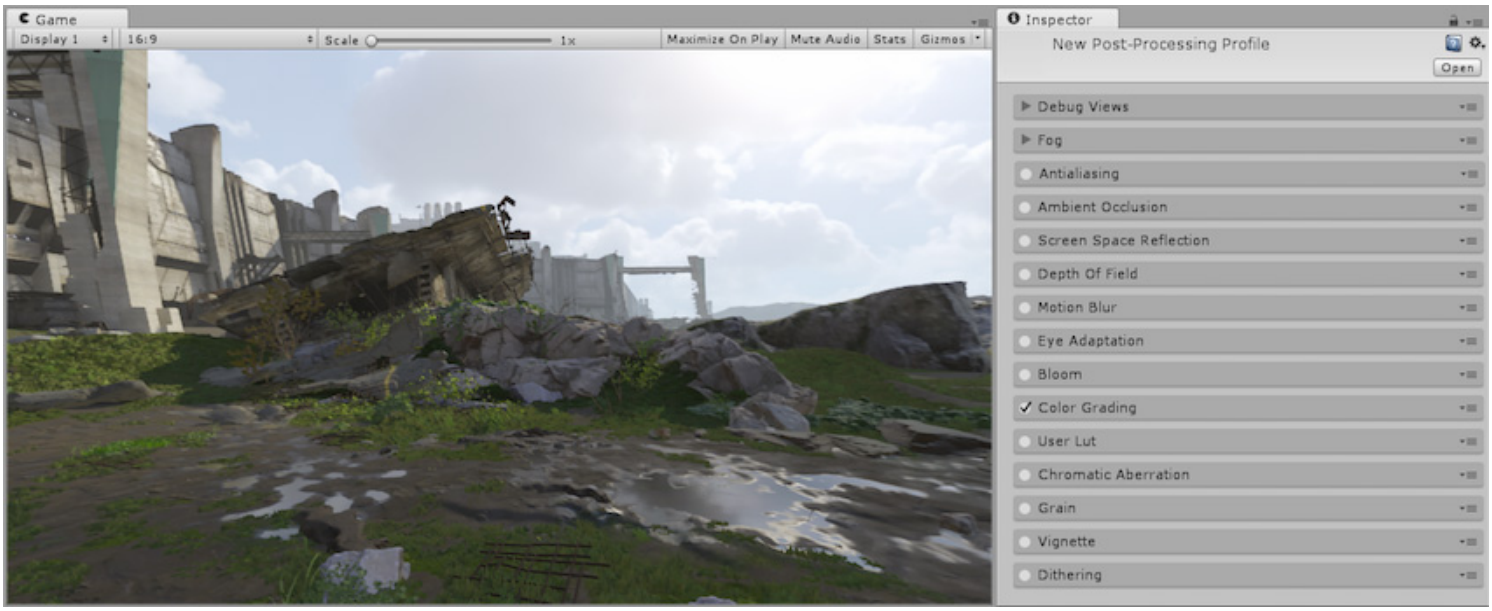
Properties

Property:	Function:
Channel	Select the output channel to modify
Red	Modify the influence of the red channel within the overall mix
Green	Modify the influence of the green channel within the overall mix
Blue	Modify the influence of the blue channel within the overall mix

Trackballs

The trackballs are used to perform three-way color grading in either Linear or Log space. When working in LDR it is recommended to use Linear trackballs for a better experience. When working in HDR it is recommended to use Log trackballs for greater control but linear trackballs can still be useful.

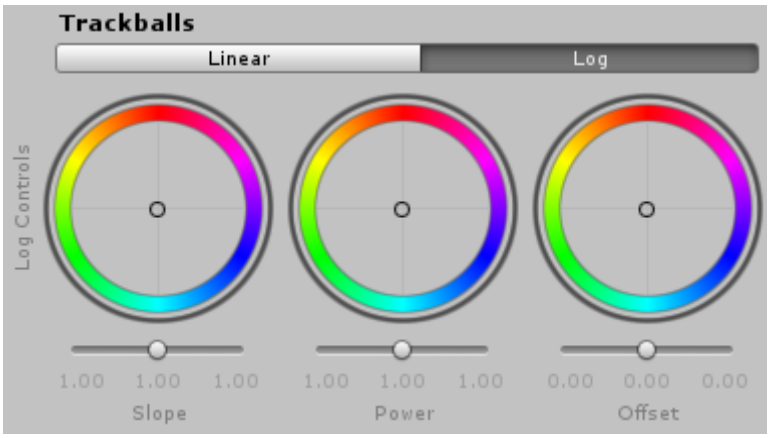
Adjusting the position of the point on the trackball will have the effect of shifting the hue of the image towards that color in the given tonal range. Different trackballs are used to affect different ranges within the image. Adjusting the slider under the trackball offsets the color lightness of that range



The same scene with only Log Trackballs applied.

Log

Log-style grading compresses the distribution of color and and contrast image data to emulate the color-timing process that could be done by optical film printers. It is generally the preferred way to do film-like grading and is highly recommended when working with HDR values.



UI for Trackballs when Log is selected

Properties

Property:	Function:
Slope	Gain function
Power	Gamma function
Offset	Shifts the entire signal

Linear

An alternative 3-way transformation to logarithmic controls optimized to work with linear-encoded data. Preferred when working in LDR.

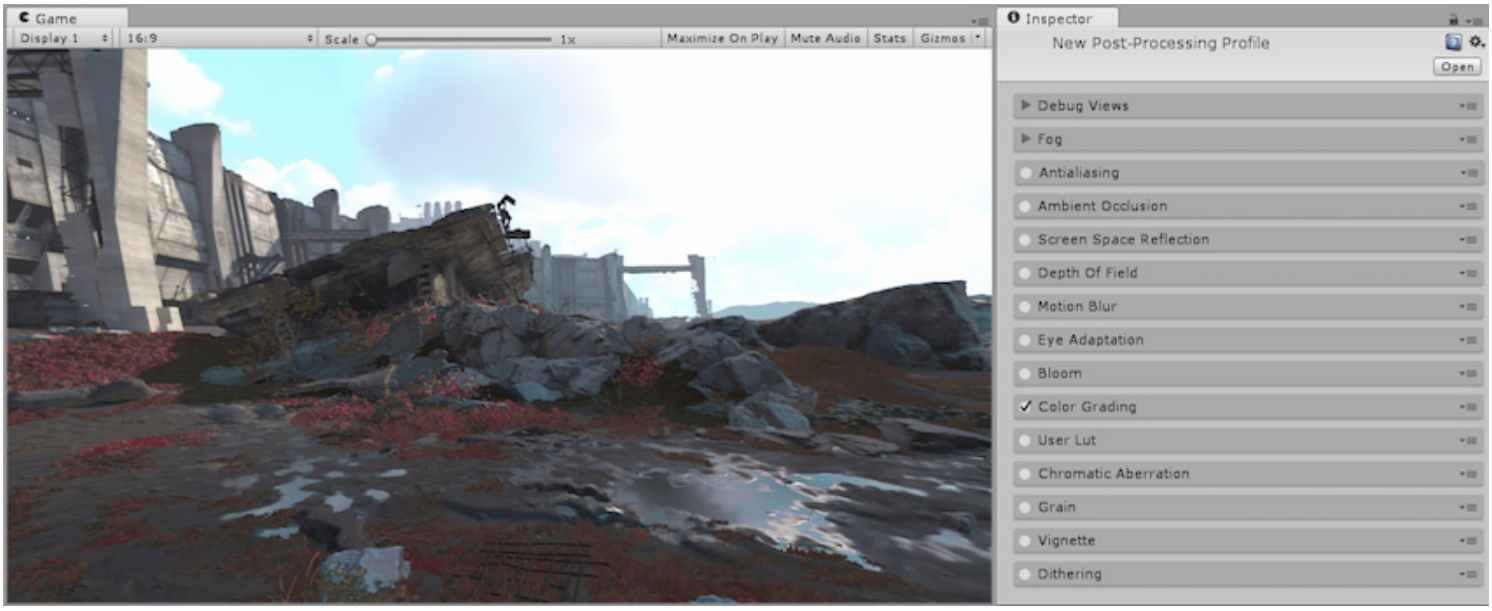
UI for Trackballs when Linear is selected

Properties

Property:	Function:
Lift	Shifts the entire signal higher or lower. Has a more pronounced effect on shadows.
Gamma	Power function that controls midrange tones.
Gain	Increases the signal. Makes highlights brighter

Grading Curves

Grading Curves (also known as versus curves) are an advanced way to adjust specific ranges in hue, saturation or luminosity in your image. By adjusting the curves on the five graphs you can achieve the effects of specific hue replacement, desaturating certain luminosities and much more.



The same scene with only Hue vs Hue Grading Curve applied to achieve color replacement

Five Grading Curve types are supplied in the post-processing stack:

YRGB

Hue vs Hue

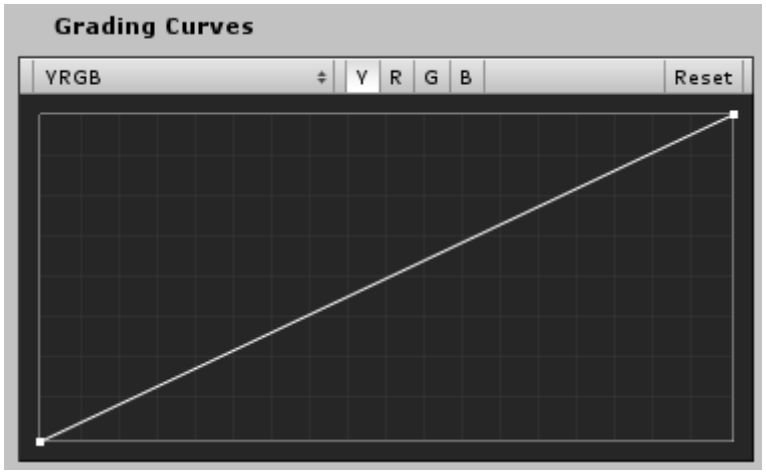
Hue vs Sat

Sat vs Sat

Lum vs Sat

YRGB Curve

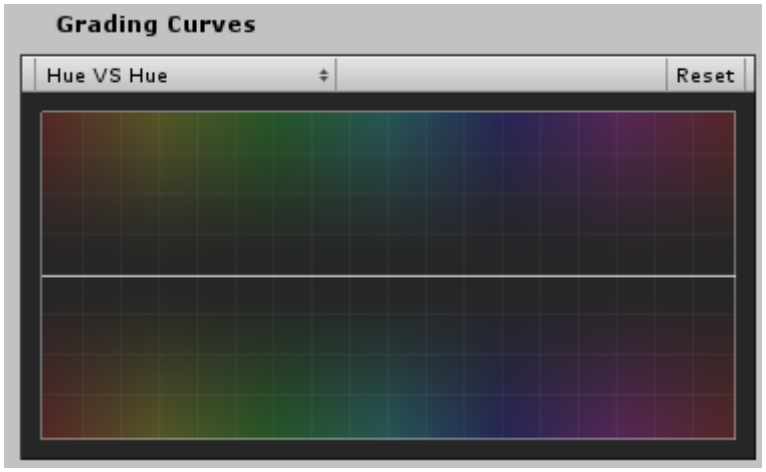
Affects the selected input channels intensity across the whole image. Input channel can be selected between Y, R, G and B where Y is a global intensity offset applied to all channels. The X axis of the graph represents input intensity and the Y axis represents output intensity. This can be used to further adjust the appearance of basic attributes such as contrast and brightness.



UI for Grading Curves when YRGB is selected

Hue vs Hue Curve

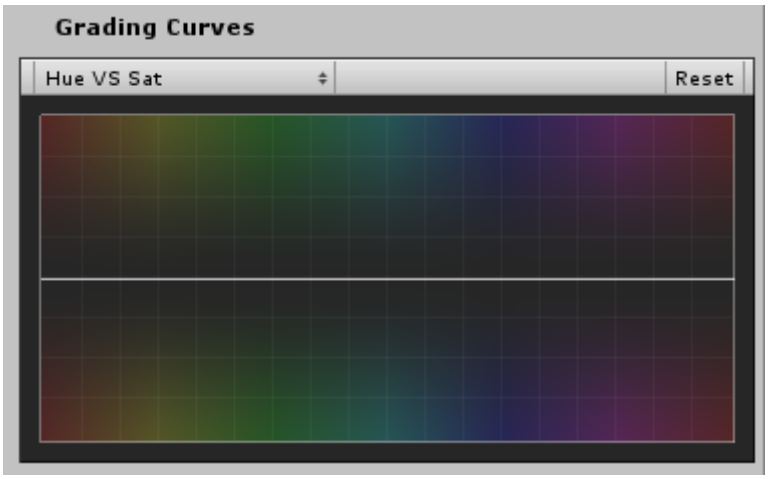
Used to shift hues within specific ranges. This curve shifts the input hue (X axis) according to the output hue (Y axis). This can be used to fine tune hues of specific ranges or perform color replacement.



UI for Grading Curves when Hue vs Hue is selected

Hue vs Sat Curve

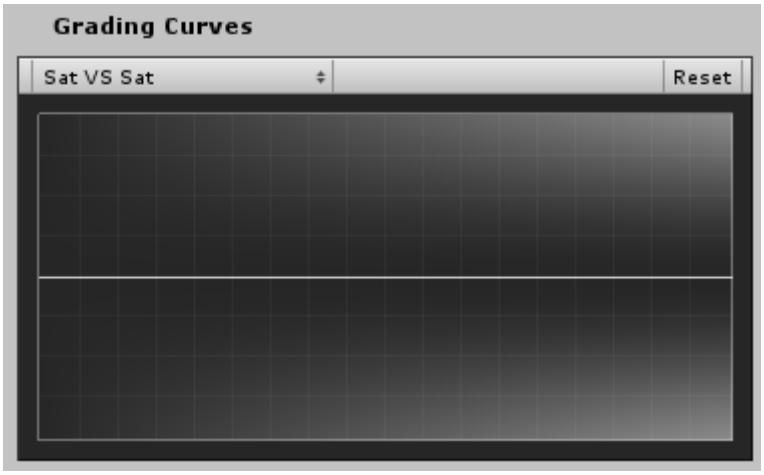
Used to adjust saturation of hues within specific ranges. This curve adjusts saturation (Y axis) according to the input hue (X axis). This can be used to tone down particularly bright areas or create artistic effects such as monochromatic except a single dominant color.



UI for Grading Curves when Hue vs Sat is selected

Sat vs Sat Curve

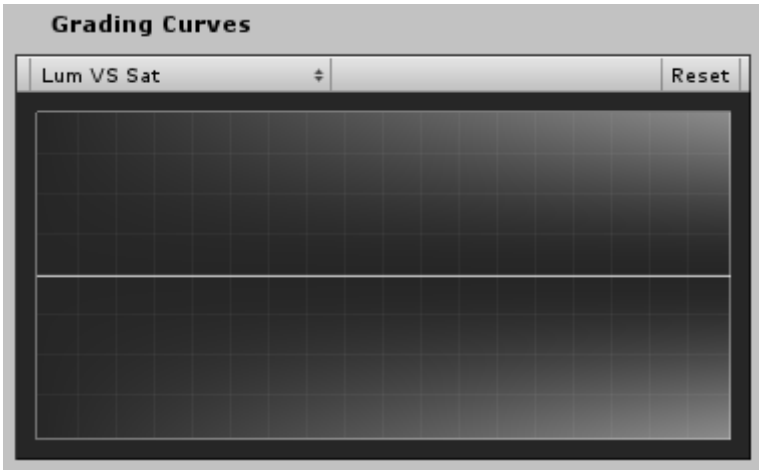
Used to adjust saturation of areas of certain saturation. This curve adjusts saturation (Y axis) according to the input saturation (X axis). This can be used to fine tune saturation adjustments made with Basic Color Grading.



UI for Grading Curves when Sat vs Sat is selected

Lum vs Sat Curve

Used to adjust saturation of areas of certain luminance. This curve adjusts saturation (Y axis) according to the input luminance (X axis). This can be used to desaturate areas of darkness to provide an interesting visual contrast.



UI for Grading Curves when Lum vs Sat is selected

2017–05–24 Page published with no editorial review

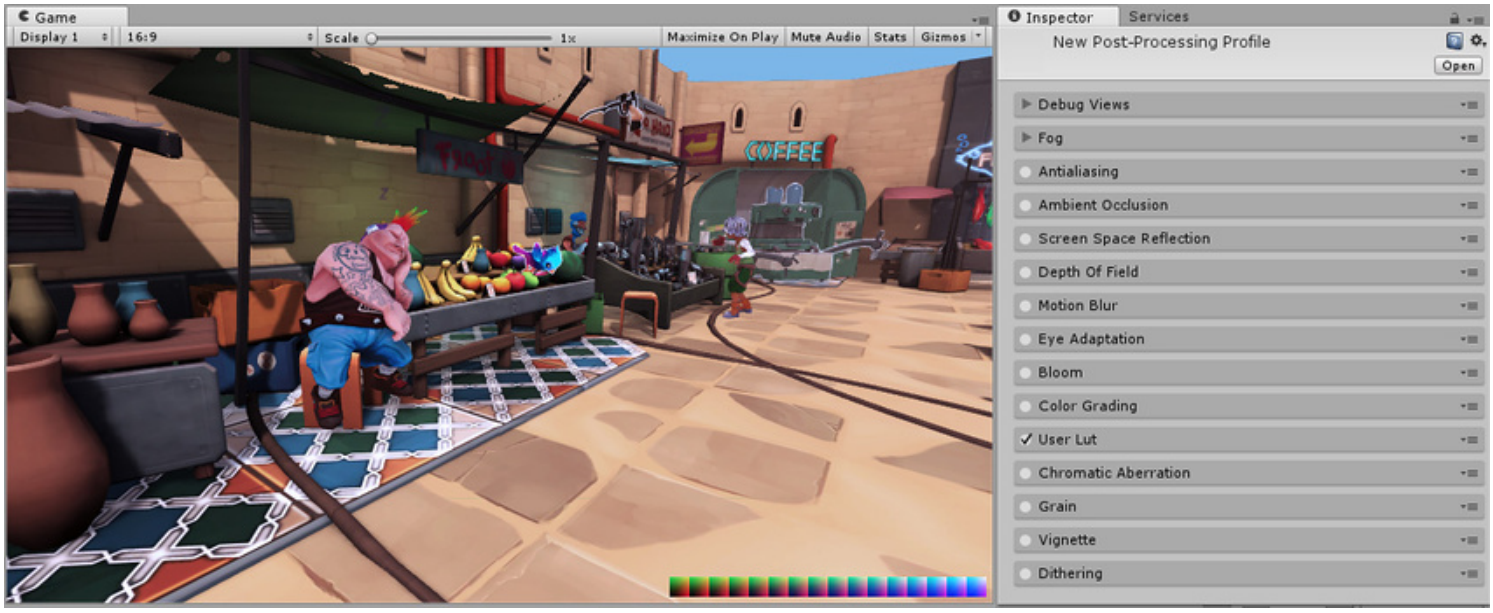
New feature in 5.6

User LUT

Leave feedback

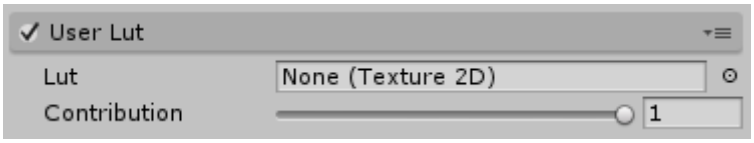
The effect descriptions on this page refer to the default effects found within the post-processing stack.

User LUT is a simpler method of color grading where **pixels** on screen are replaced by new values from an LUT (or look-up texture) supplied by the user. It is a much less advanced method than the Color Grading effect. However, as this method does not require the more advanced **texture formats** used by Color Grading it is recommended as a fallback for platforms that do not support these formats.



Scene with User LUT (LUT overlaid for demonstrative purposes)

Scene without User LUT (LUT overlaid for demonstrative purposes)



UI for User LUT

Properties

Property:	Function:

Property:	Function:
Lut	Custom lookup texture (strip format, e.g. 256x16).
Contribution	Blending factor.

Optimisation

If using a 1024x32 texture for input, consider using a 256x16 instead

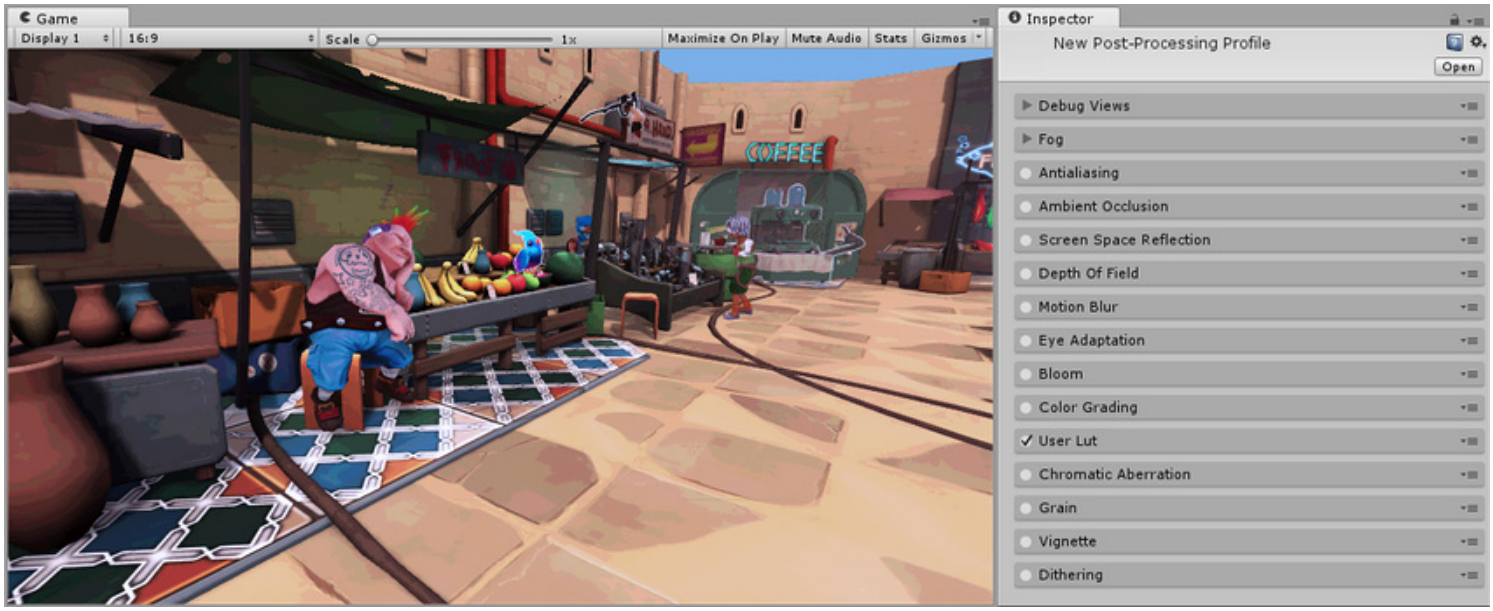
Details

User LUT uses a “strip format” texture for input. Two neutral LUTs are provided with the Post-processing stack, one at a resolution of 256x16 and another at 1024x32. Using larger input textures will affect performance.

To create an LUT import one of the neutral LUTs into an image editing tool such as Photoshop with a screenshot of your **scene**. Apply color corrections in a non destructive manner on top of these two images until you are happy with the result. Note that only pixel-local effects are supported by LUTs, meaning no blur and other effects that depends on the value of neighboring pixels. Now export the LUT with these color changes applied back into Unity to be used in the User LUT effect.

The User LUT effect will prompt you to make changes to the texture’s import settings if necessary.

You can achieve a “lo-fi” effect by manually setting the **Filter Mode** of the input texture to **Point (no filter)**.



Scene using the same LUT as above, but with Filter Mode set to Point

Requirements

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

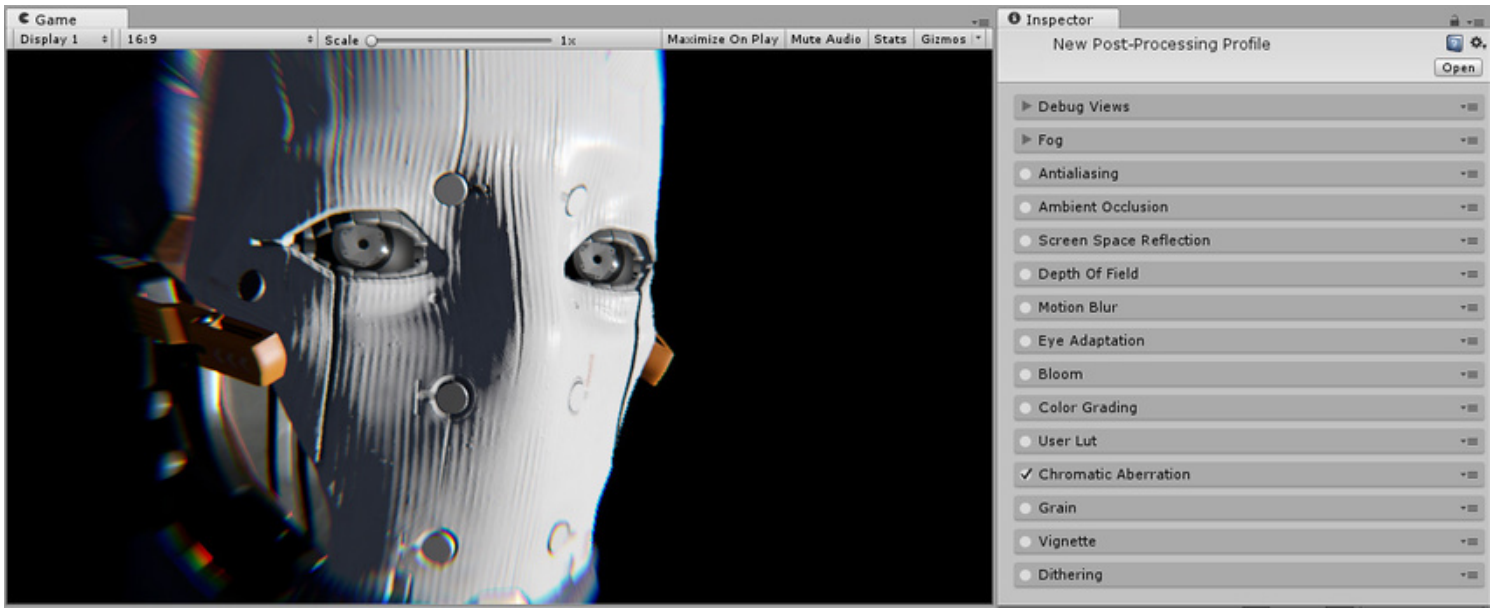
Chromatic Aberration

Leave feedback

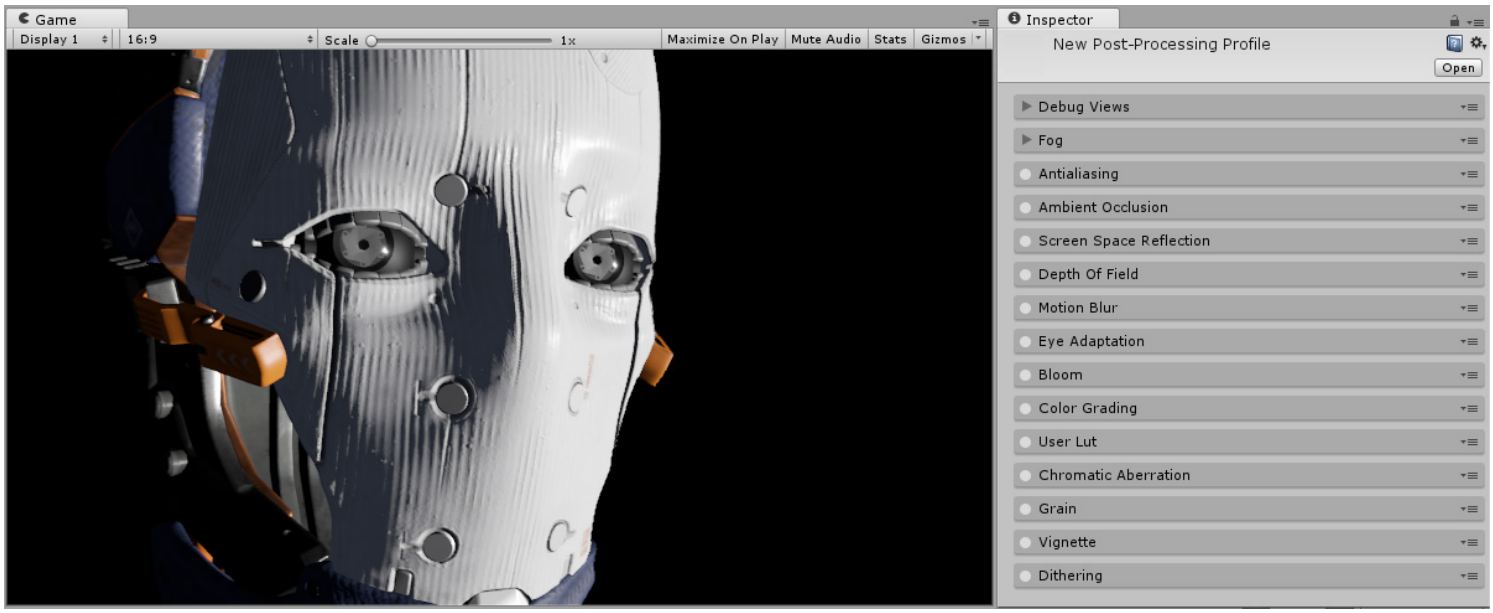
The effect descriptions on this page refer to the default effects found within the post-processing stack.

In photography, chromatic aberration is an effect resulting from a **camera**'s lens failing to converge all colors to the same point. It appears as "fringes" of color along boundaries that separate dark and bright parts of the image.

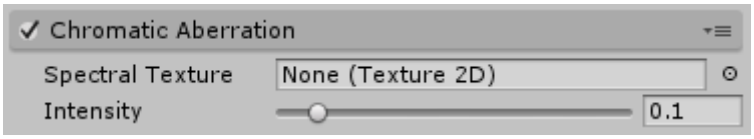
The Chromatic Aberration effect is used to replicate this camera defect, it is also often used to artistic effect such as part of camera impact or intoxication effects. This implementation provides support for red/blue and green/purple fringing as well as user defined color fringing via an input texture.



Scene with Chromatic Aberration



Scene without Chromatic Aberration



UI for Chromatic Aberration

Properties

Property:	Function:
Intensity	Strength of chromatic aberrations.
Spectral Texture	Texture used for custom fringing color (will use default when empty)

Optimisation

Reduce **Intensity**

Details

Performances depend on the **Intensity** value (the higher it is, the slower the render will be as it will need more samples to render smooth chromatic aberrations).

Chromatic Aberration uses a **Spectral Texture** input for custom fringing. Four example spectral textures are provided with the Post-processing stack:

Red/Blue (Default)

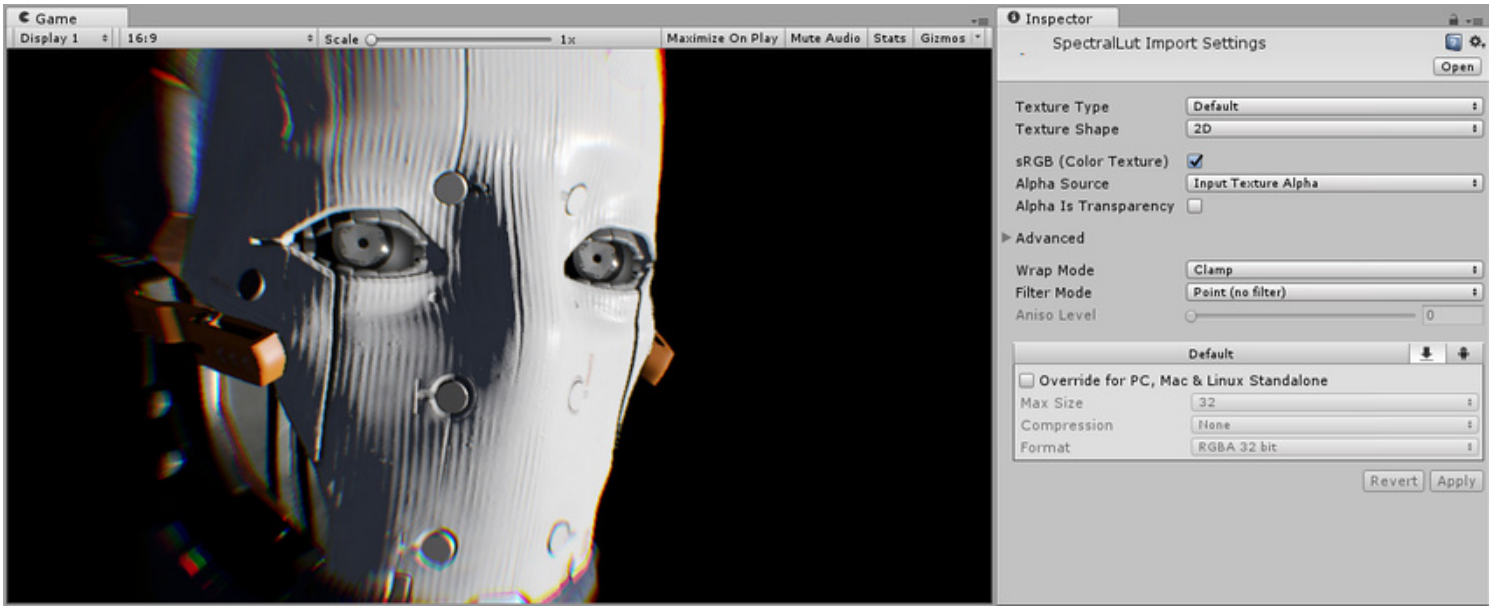
Blue/Red

Green/Purple

Purple/Green

You can create custom spectral textures in any image editing software. **Spectral Texture** resolution is not constrained but it is recommended that they are as small as possible (such as the 3x1 textures provided).

You can achieve a less smooth effect by manually setting the **Filter Mode** of the input texture to **Point (no filter)**.



Scene using the same Chromatic Aberration as above, but with Filter Mode set to Point

Requirements

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

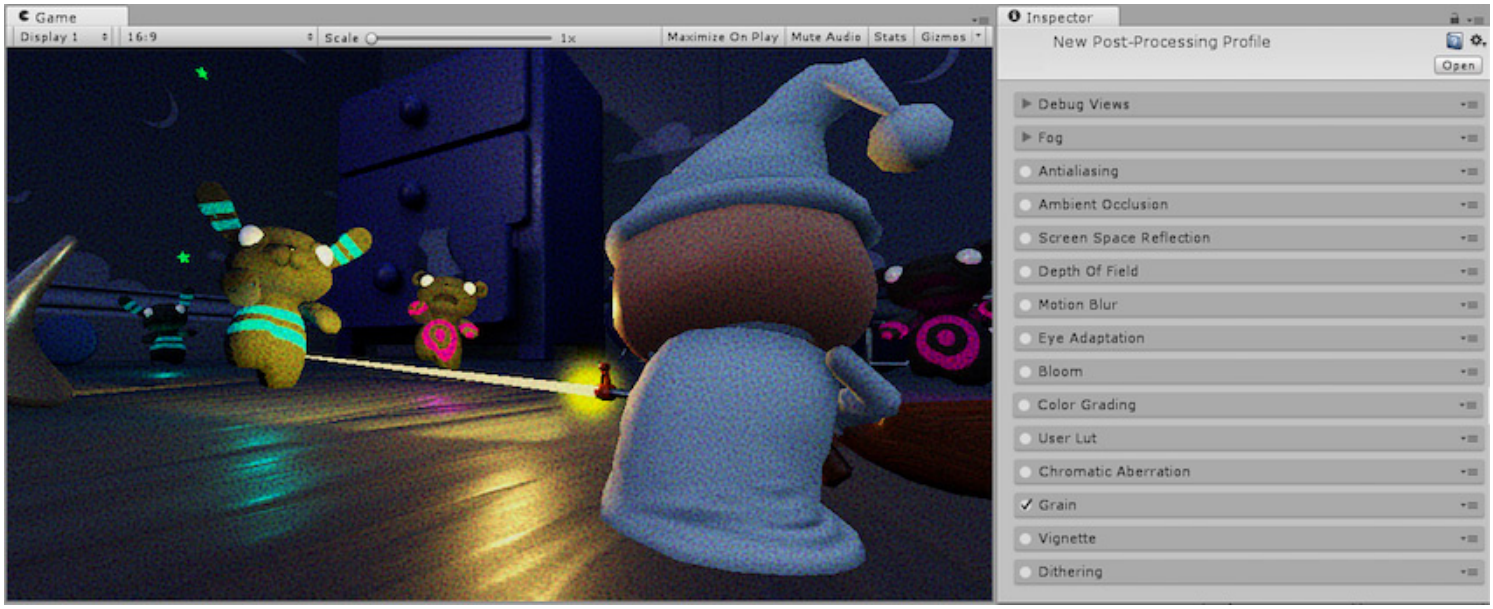
Grain

Leave feedback

The effect descriptions on this page refer to the default effects found within the post-processing stack.

Film grain is the random optical texture of photographic film due to the presence of small particles in the metallic silver of the film stock.

The Grain effect in the post-processing stack is based on a coherent gradient noise. It is commonly used to emulate the apparent imperfections of film and often exaggerated in horror themed games.



Scene with Grain

Scene without Grain

UI for Grain

Properties

Property:	Function:
Intensity	Grain strength. Higher means more visible grain.
Luminance Contribution	Controls the noisiness response curve based on **scene** luminance. Lower values mean less noise in dark areas.
Size	Grain particle size.
Colored	Enable the use of colored grain.

Optimisation

Disabled Colored

Requirements

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

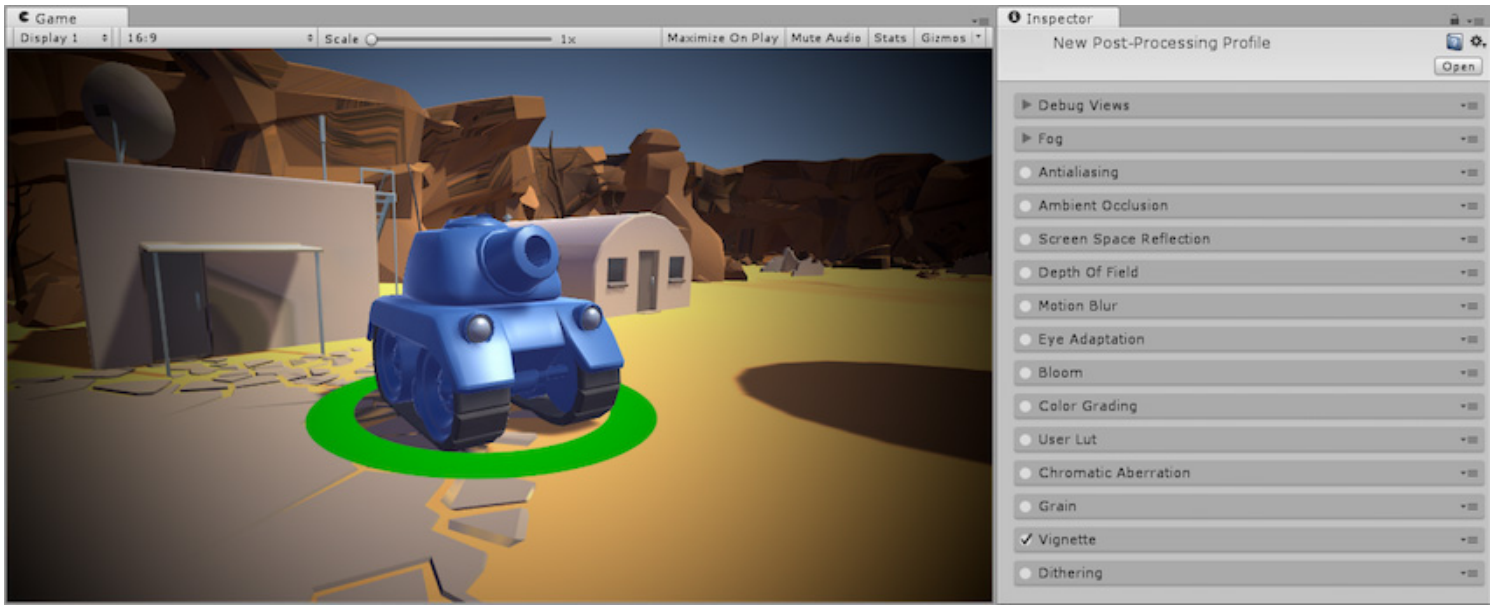
New feature in 5.6

Vignette

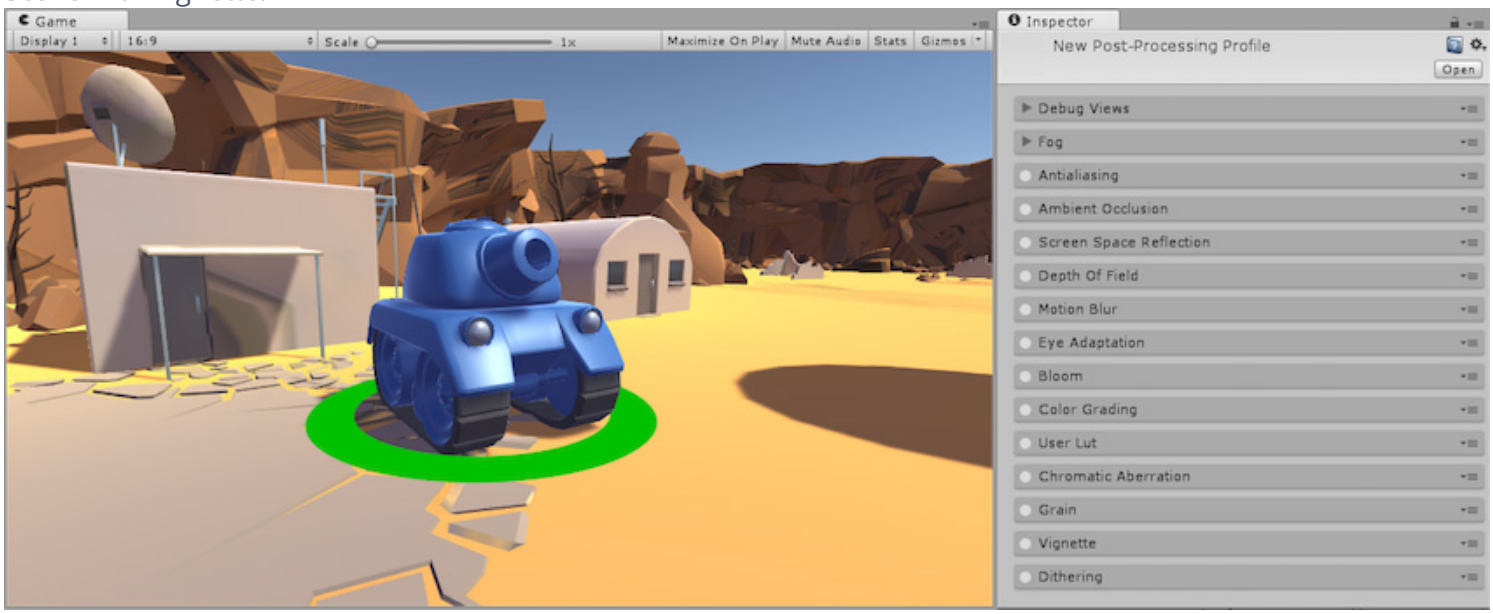
Leave feedback

The effect descriptions on this page refer to the default effects found within the post-processing stack.

In Photography, vignetting is the term used for the darkening and/or desaturating towards the edges of an image compared to the center. This is usually caused by thick or stacked filters, secondary lenses, and improper lens hoods. It is also often used for artistic effect, such as to draw focus to the center of an image.



Scene with Vignette.



Scene without Vignette.

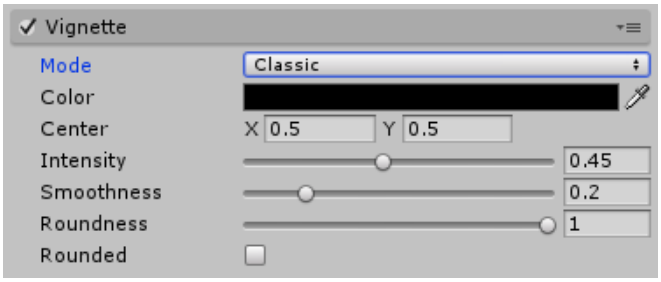
The Vignette effect in the post-processing stack comes in 2 modes

Classic

Masked

Classic

Classic mode offers parametric controls for the position, shape and intensity of the Vignette. This is the most common way to use the effect.



UI for Vignette when Classic is selected

Properties

Property:	Function:
Color	Vignette color. Use the alpha channel for transparency.
Center	Sets the vignette center point (screen center is [0.5,0.5]).
Intensity	Amount of vignetting on screen.
Smoothness	Smoothness of the vignette borders.
Roundness	Lower values will make a more squared vignette.
Rounded	Should the vignette be perfectly round or be dependent on the current **aspect ratio**?

Optimisation

N/A

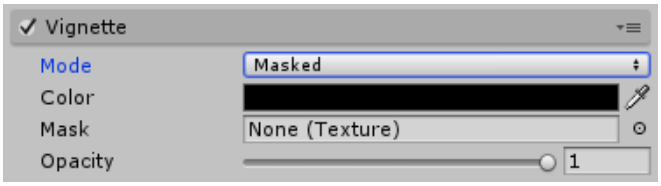
Requirements

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

Masked

Masked mode multiplies a custom texture mask over the screen to create a Vignette effect. This mode can be used to achieve less common vignetting effects.



UI for Vignette when Masked is selected

Properties

Property:	Function:
Color	Vignette color. Use the alpha channel for transparency.
Mask	A black and white mask to use as a vignette.
Intensity	Mask opacity.

Optimisation

N/A

Requirements

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

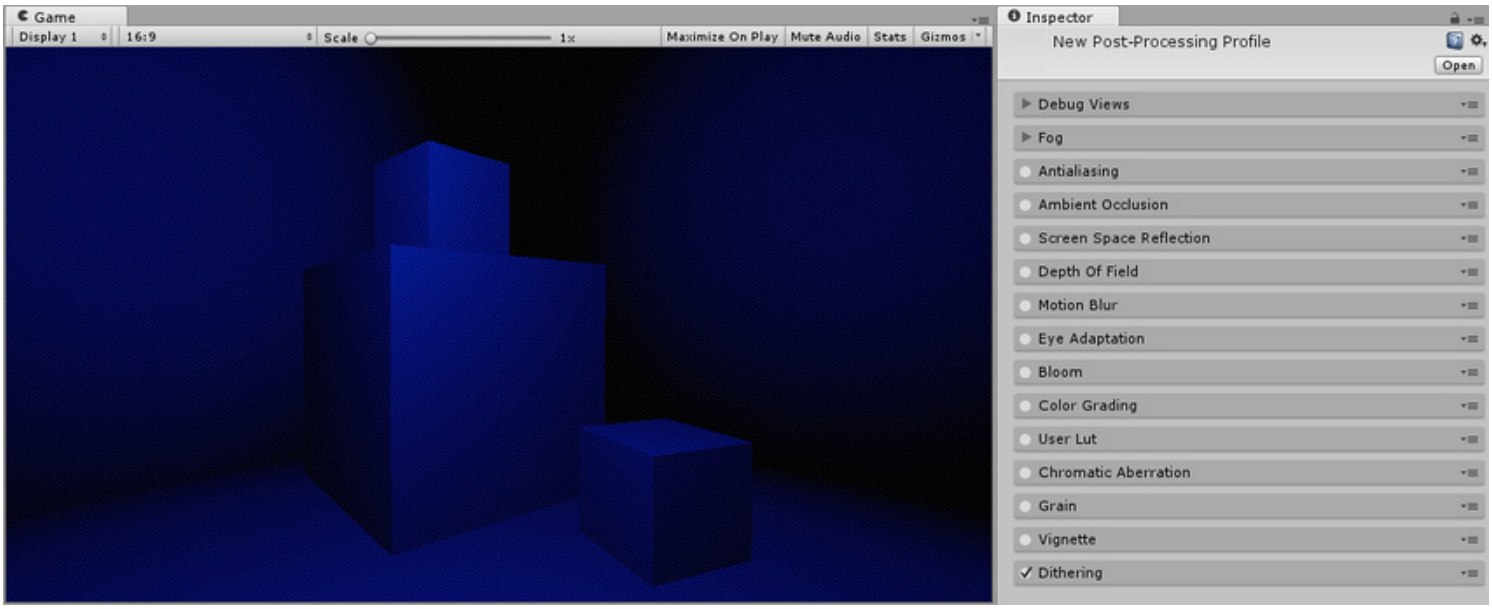
New feature in 5.6

Dithering

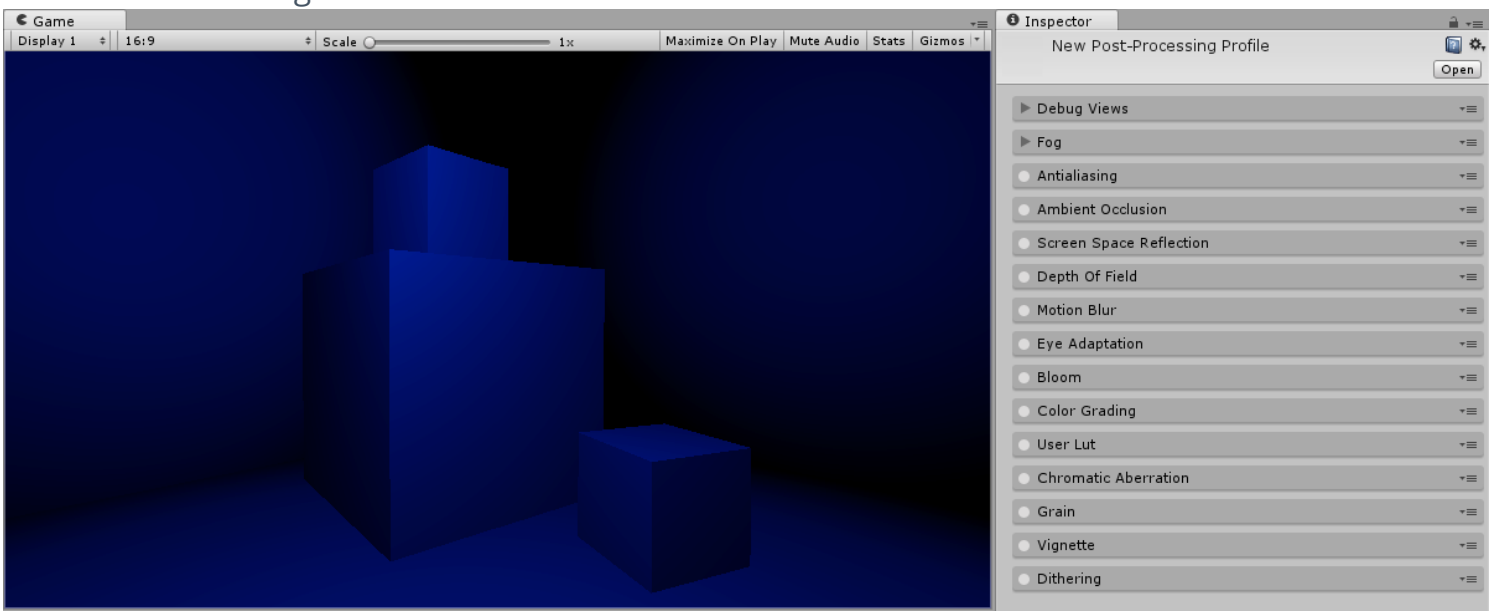
Leave feedback

The effect descriptions on this page refer to the default effects found within the post-processing stack.

Dithering is the process of intentionally applying noise as to randomize quantization error. This prevents large-scale patterns such as color banding in images.



Scene with Dithering



Scene without Dithering

Requirements

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

Debug Views

Leave feedback

The post-processing stack comes with a selection of debug views to view specific effects or passes such as depth, normals and motion vectors. These can be found at the top of the post-processing profile. Debug views will affect the application of other effects on the Game view.

The debug views included in the post-processing stack are:

Depth

Normals

Motion Vectors

Ambient Occlusion

Eye Adaptation

Focus Plane

Pre Grading Log

Log Lut

User Lut

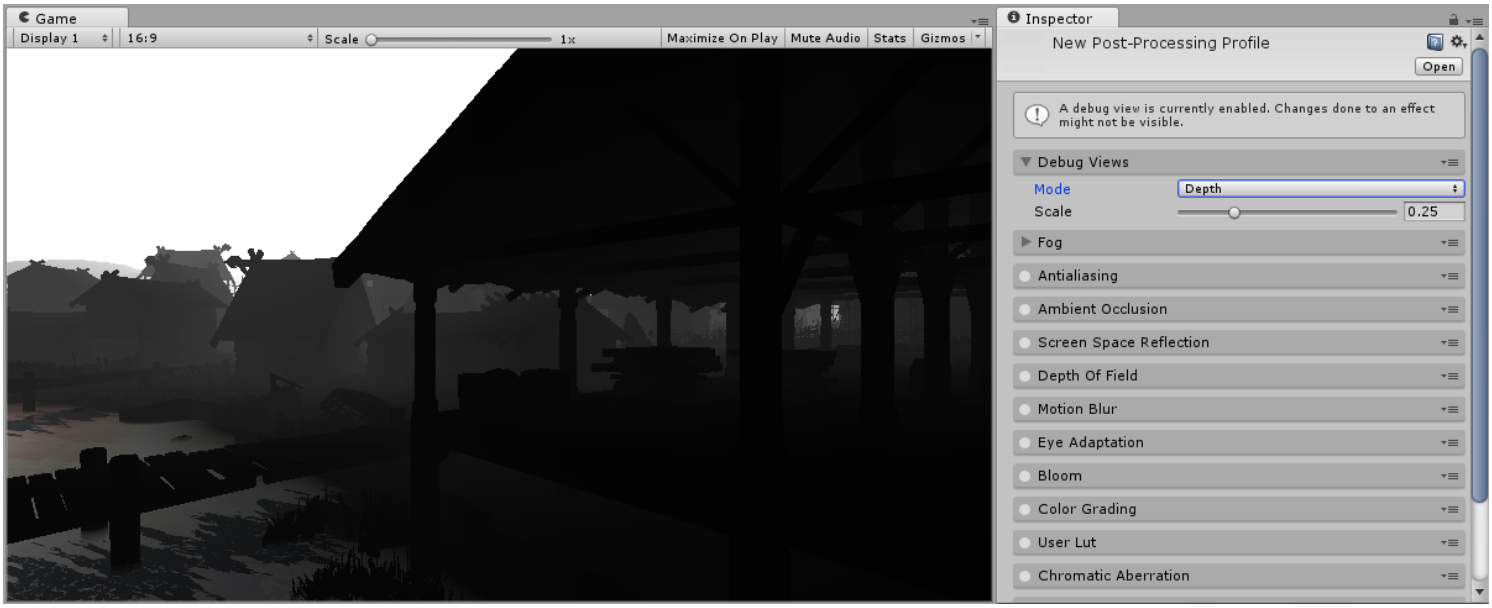
Properties

Property:	Function:
Mode	The currently selected debug view

Depth

The Depth Debug View displays the depth values of the **pixels** on screen. These values can be shifted for easier viewing by adjusting the Scale value.

The Debug View can be used to identify issues in all effects that use the depth texture, such as Ambient Occlusion and Depth of Field.



The Depth Debug View

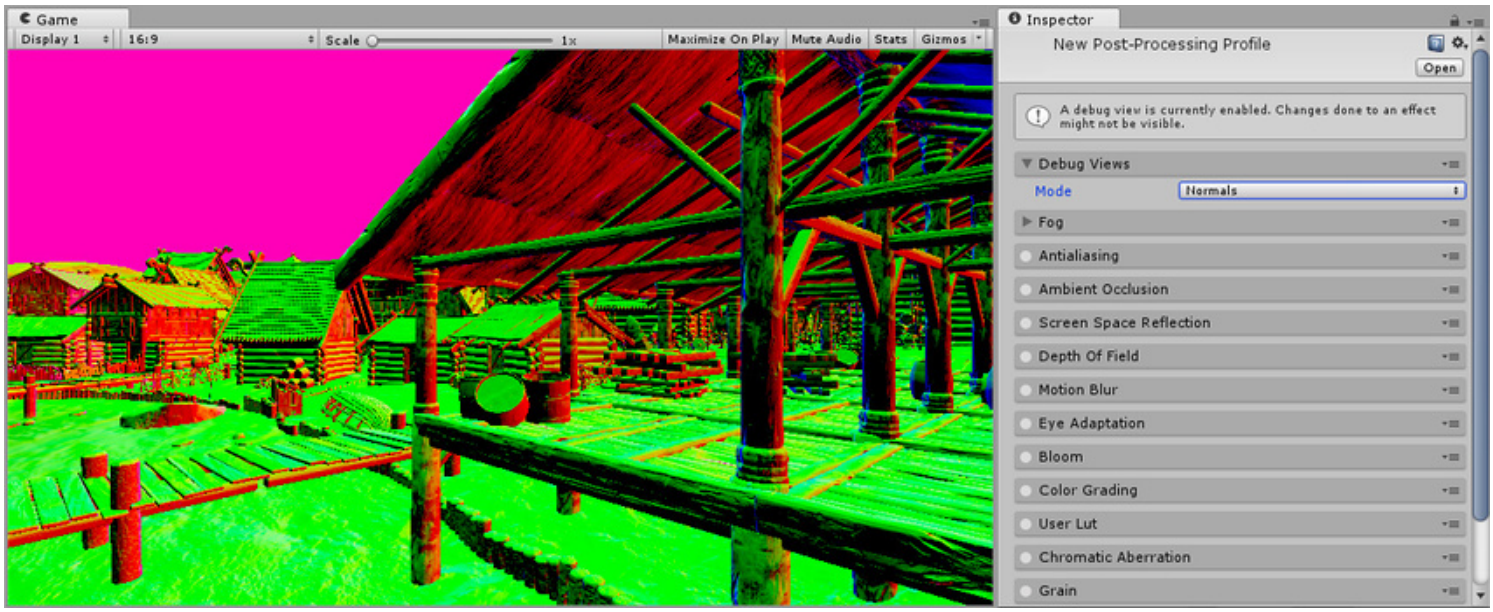
Properties

Property:	Function:
Scale	Scales the **camera** far plane before displaying the depth map.

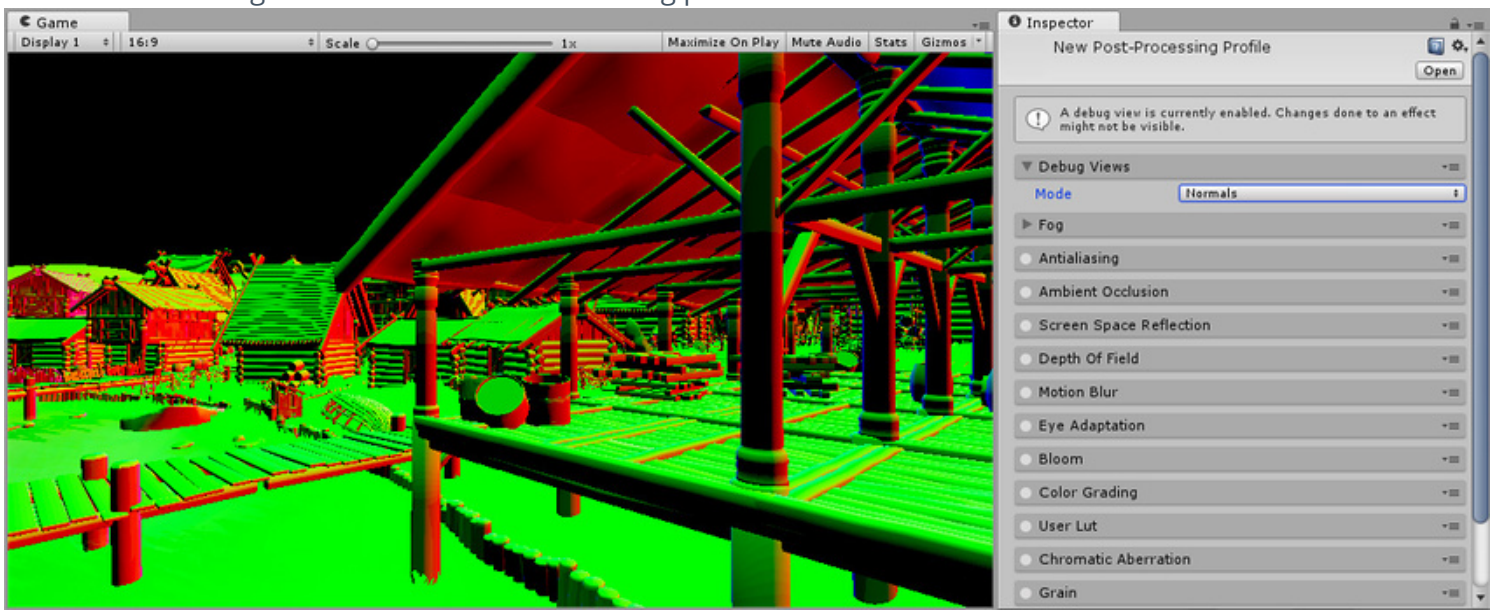
Normals

The Normals Debug View displays the normals texture used for various effects. This Debug View differs between **rendering** paths. In the Deferred rendering path it displays the G-Buffer Normals texture, this includes details from object's normal maps. In the **Forward rendering** path it just displays object's vertex normals.

This Debug View can be used to identify issues in all effects that use the normals texture such as Screen Space Reflection and Ambient Occlusion.



The Normals Debug View in the Deferred rendering path



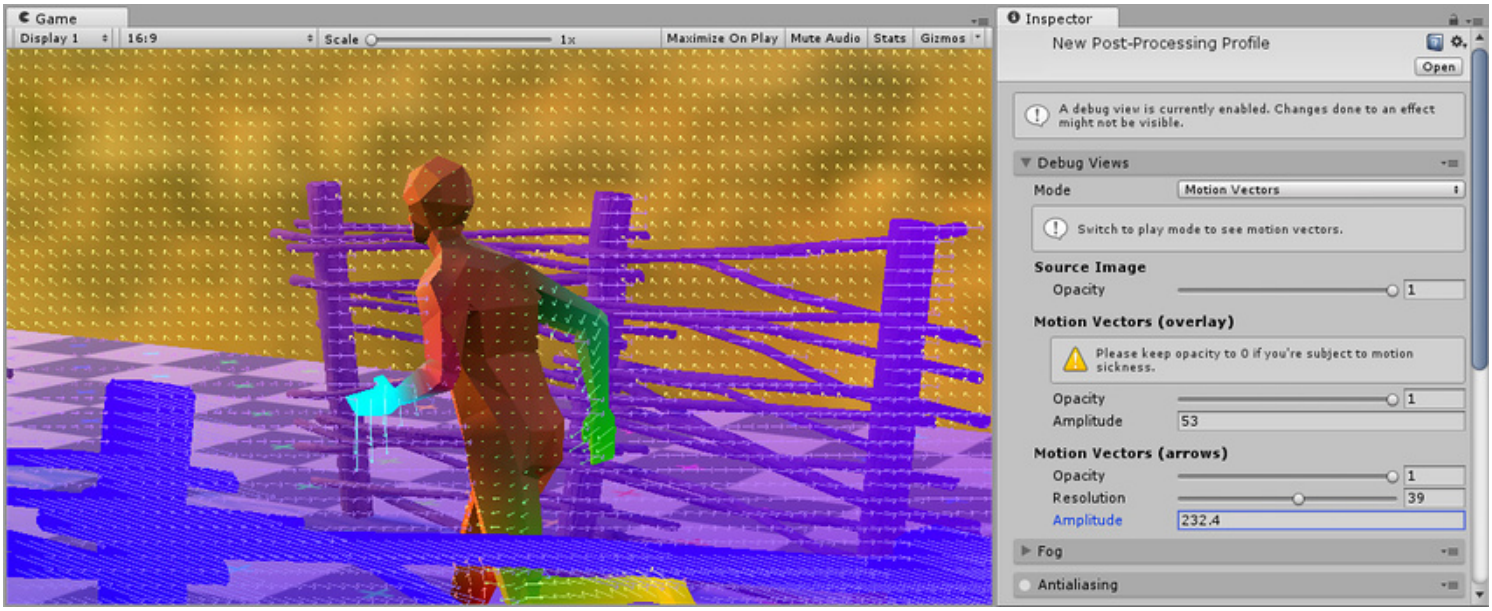
The Normals Debug View in the Forward rendering path

Motion Vectors

The Motion Vectors Debug View displays visualizations of the motion vector texture. There are two types of visualizations in the Motion Vectors Debug View.

The overlay visualization displays motion vector colors of each pixel on screen. Different colors show different movement directions and more saturated values indicate higher velocities. The arrows visualization draws arrows on screen indicating direction and velocity of movement, this is less precise but easier to read.

This Debug View can be used to identify issues with temporal effects such as Motion Blur and Temporal Anti-aliasing.



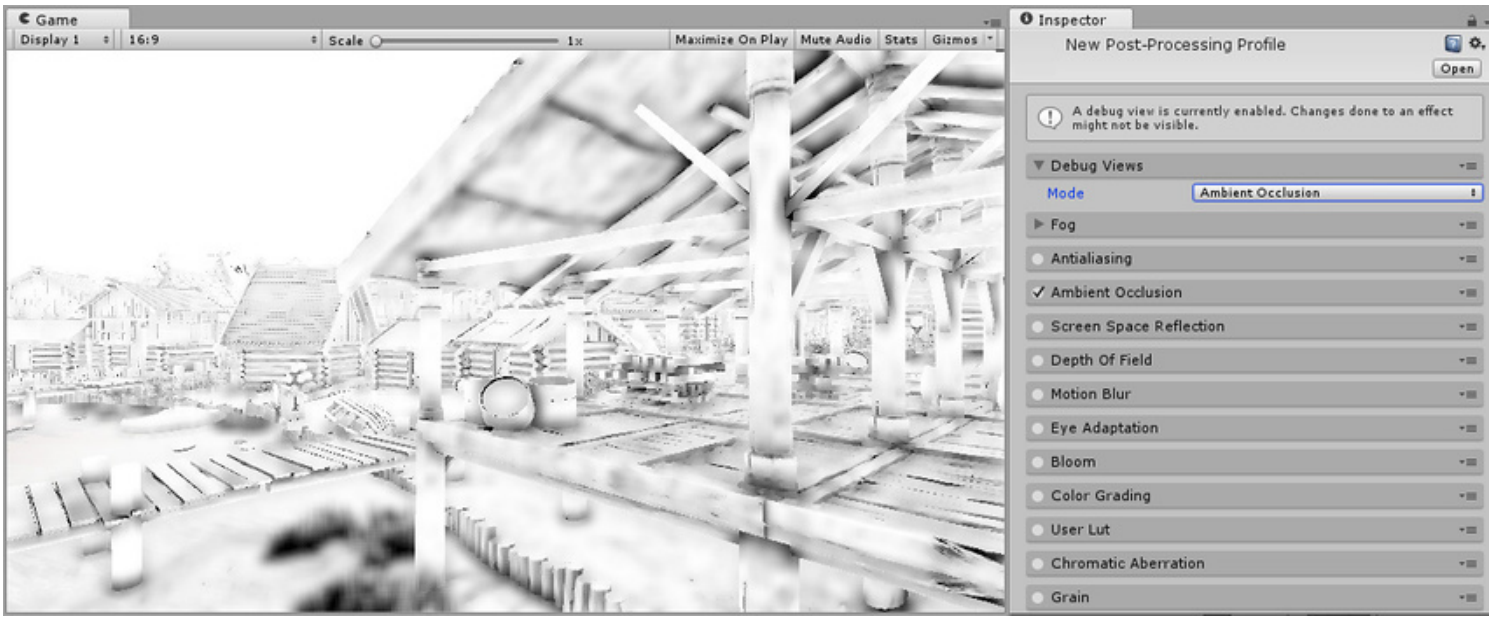
The Motion Vectors Debug View

Properties

Property:	Function:
(Source Image) Opacity	Opacity of the source render.
(Motion Vectors (overlay)) Opacity	Opacity of the per-pixel motion vectors
(Motion Vectors (overlay)) Amplitude	Because motion vectors are mainly very small vectors, you can use this setting to make them more visible.
(Motion Vectors (arrows)) Opacity	Opacity for the motion vector arrows.
(Motion Vectors (arrows)) Resolution	The arrow density on screen.
(Motion Vectors (arrows)) Amplitude	Tweaks the arrows length.

Ambient Occlusion

The Ambient Occlusion Debug View displays the final result of the Ambient Occlusion effect without multiplying it on top of the **scene**. This can be useful for identifying problems in Ambient Occlusion.

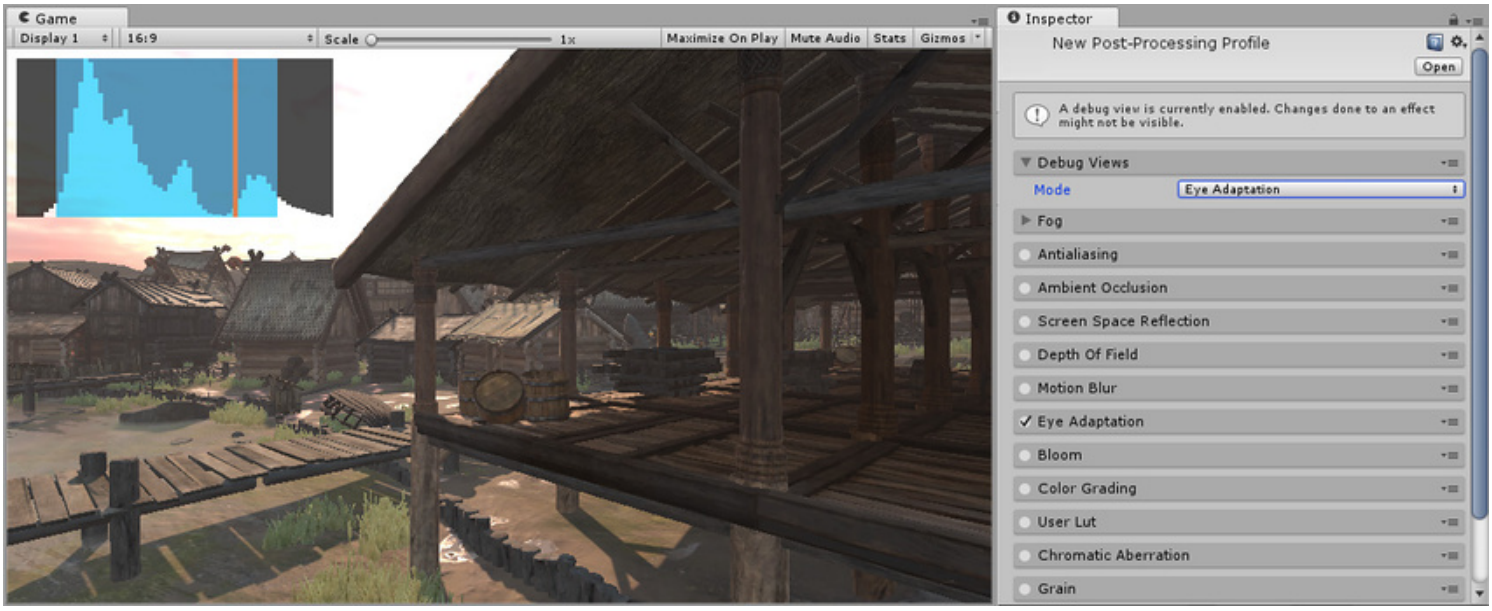


The Ambient Occlusion Debug View.

Eye Adaptation

The Eye Adaptation Debug View displays a representation of the histogram used for Eye Adaptation. This indicates the minimum and maximum for Histogram log and Luminance as well as the current average luminance computed by the effect. These are overlaid onto the luminance values of the screen, updated in real time to indicate the Eye Adaptation effect.

This is useful for fine tuning the Eye Adaptation effect.

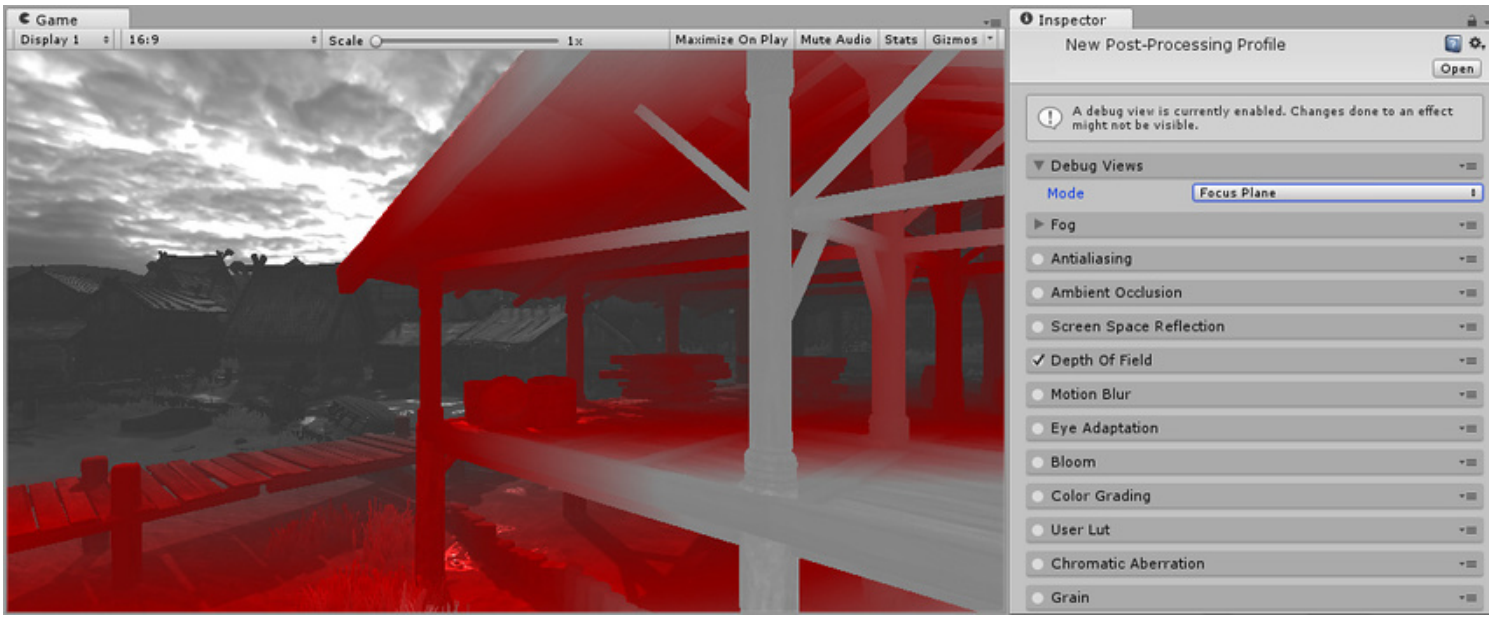


The Eye Adaptation Debug View.

Focus Plane

The Focus Plane Debug View displays the focus distance and aperture range used for Depth of Field.

This is useful for setting up the Depth of Field effect.

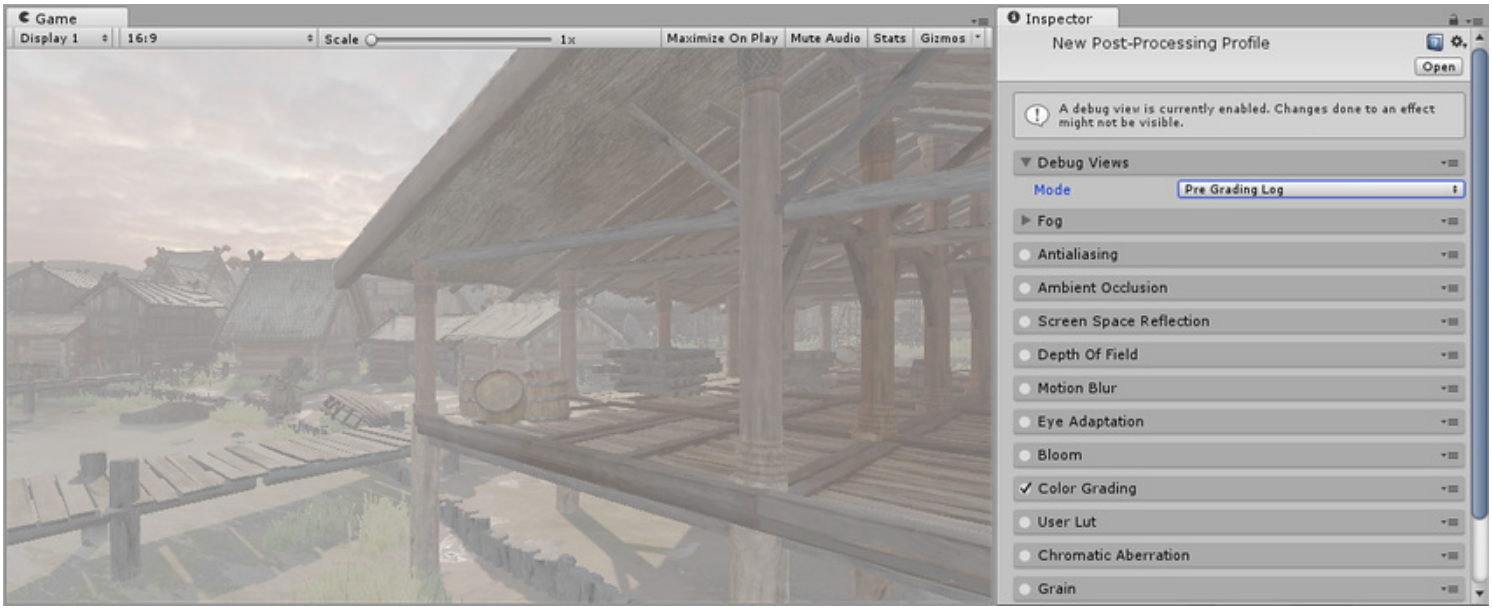


The Focus Plane Debug View.

Pre Grading Log

The Pre Grading Log Debug View displays the source image in Log space. This is the input used for most controls of Color Grading. It displays a log/compressed view of the current **HDR** view.

This is useful for identifying problems in Color Grading.

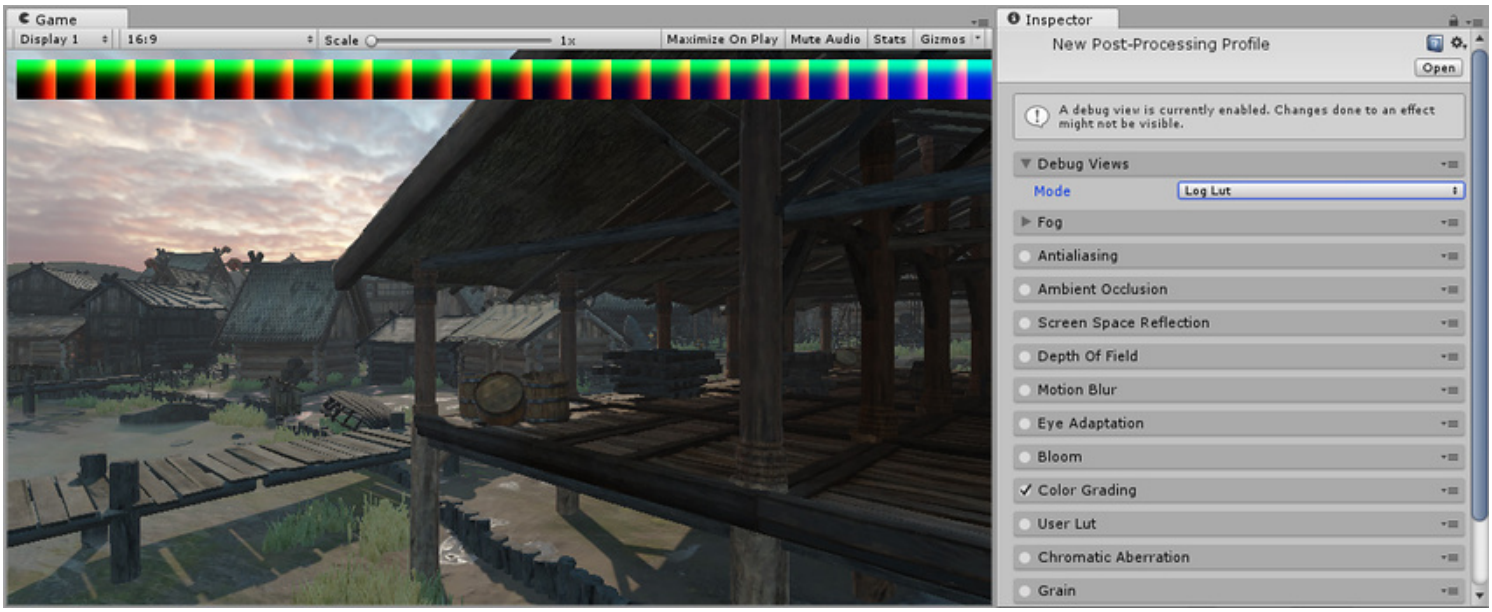


The Pre Grading Log Debug View.

Log Lut

The Log Lut Debug View displays the output Lut for Color Grading. This is the Lut generated by the Color Grading effect based on parameters set by the user.

This is useful for identifying problems in Color Grading.

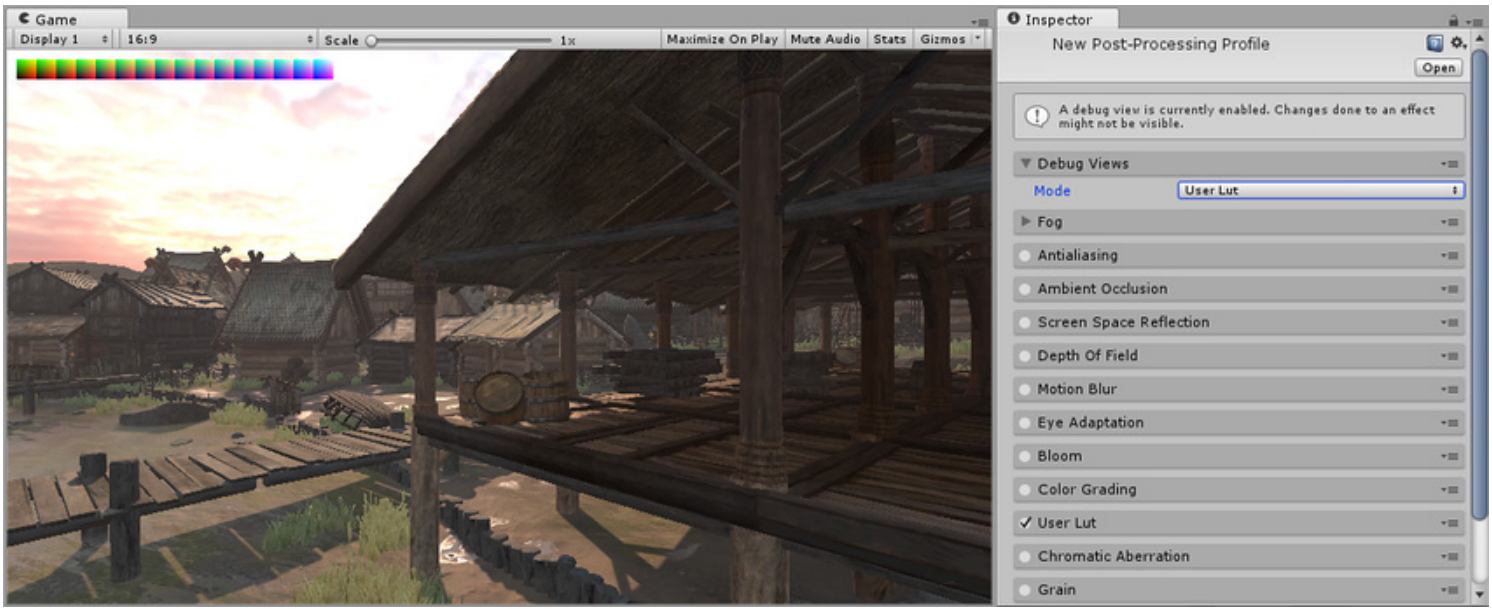


The Log Lut Debug View.

User Lut

The User Lut Debug View displays the output Lut for User Lut. This is the texture set in the Lut field adjusted by the **Contribution** parameter.

This is useful for identifying problems in User Lut.



The User Lut Debug View.

2017–05–24 Page published with no editorial review

New feature in 5.6

Monitors

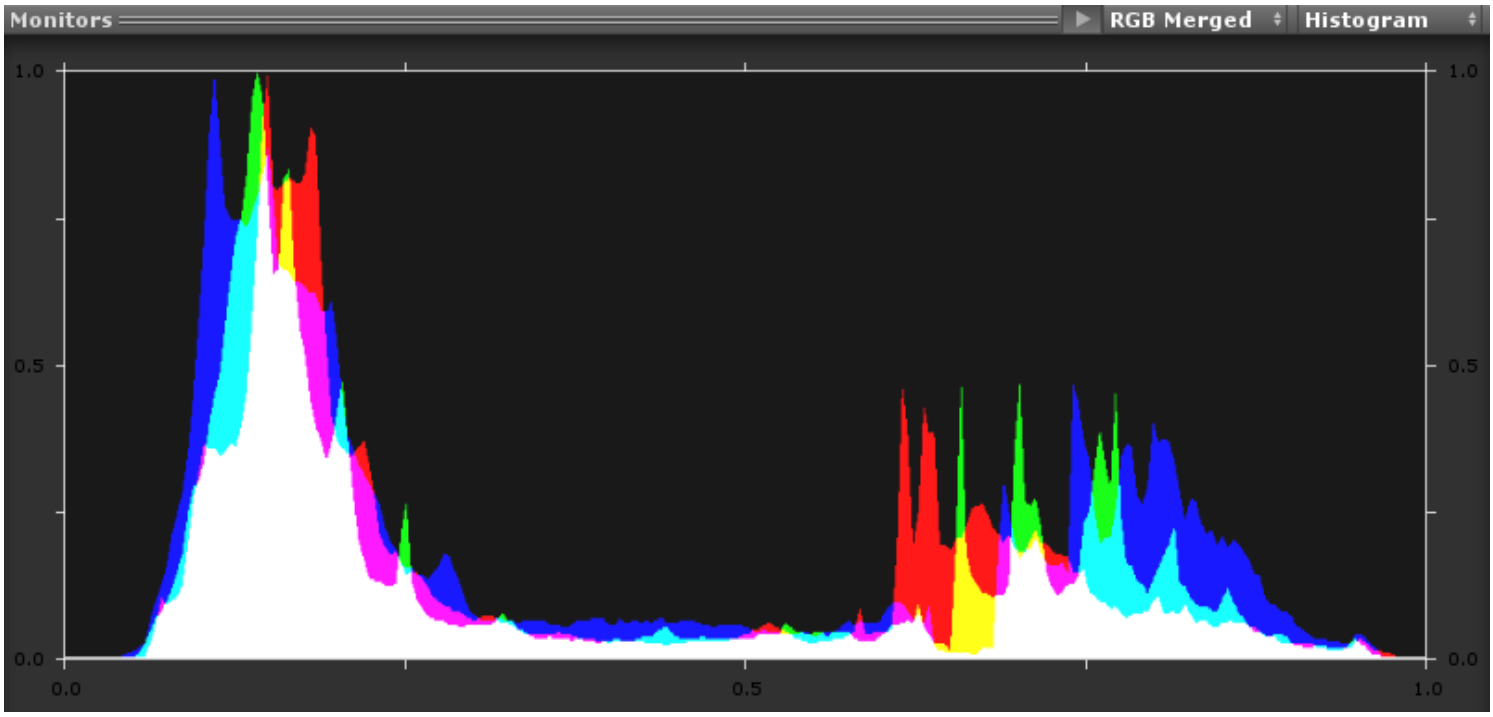
Leave feedback

To help artists control the overall look and exposure of a **scene**, the post-processing stack comes with a set of industry-standard monitors. You can find them in the preview area of the **Inspector**. Like with any other preview area in the **Inspector**, you can show/hide it by clicking it and undock it by right-clicking its title.

Each monitor can be enabled in play mode for real-time update by clicking the button with the play icon in the titlebar. Note that this can greatly affect performance of your scene in the editor, so use with caution. This feature is only available on compute-shader enabled platforms.

Histogram

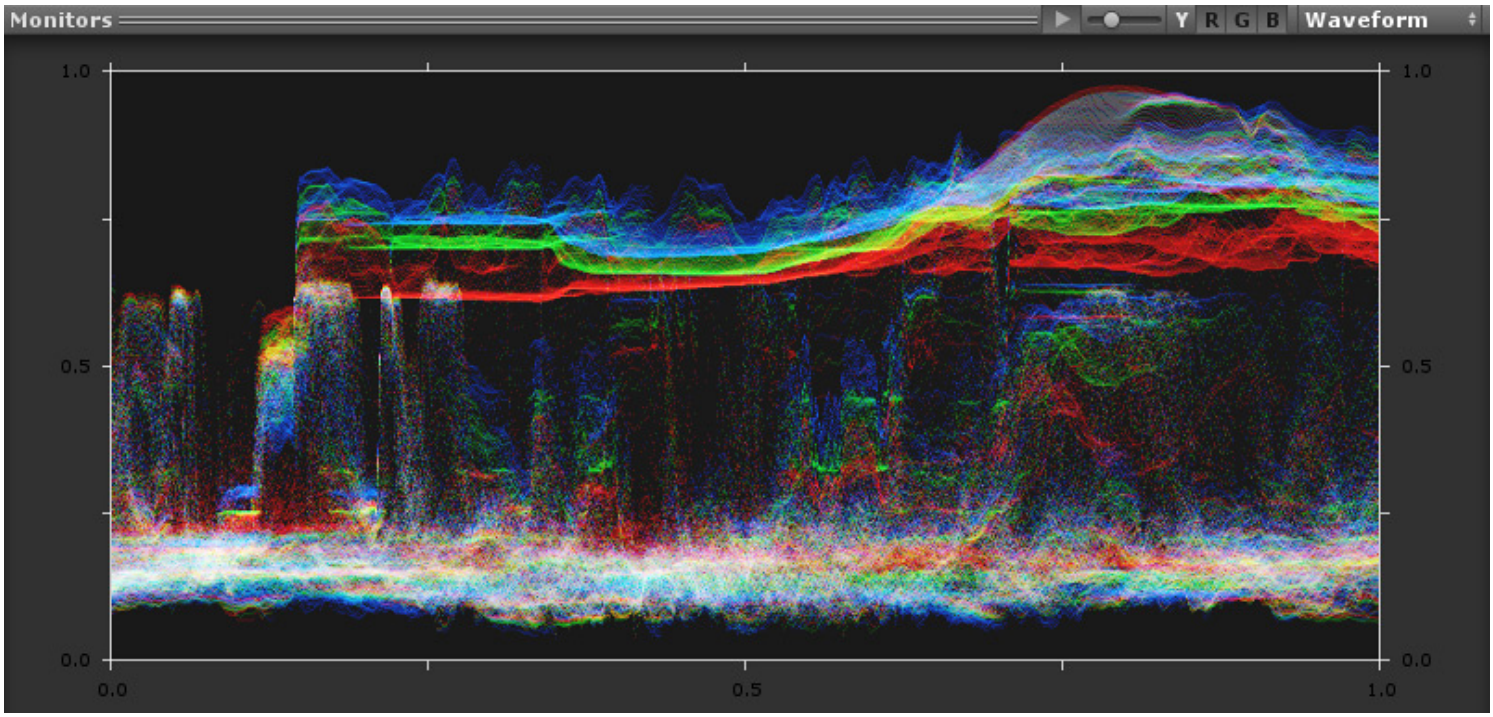
A standard gamma histogram, similar to the one found in common graphics editing software. A histogram illustrates how **pixels** in an image are distributed by graphing the number of **pixels** at each color intensity level. It can help you determine whether an image is properly exposed or not.



The Histogram Monitor

Waveform

This monitors displays the full range of luma information in the render. The horizontal axis of the graph corresponds to the render (from left to right) and the vertical axis is the luminance. You can think of it as an advanced histogram, with one vertical histogram for each column of your image.

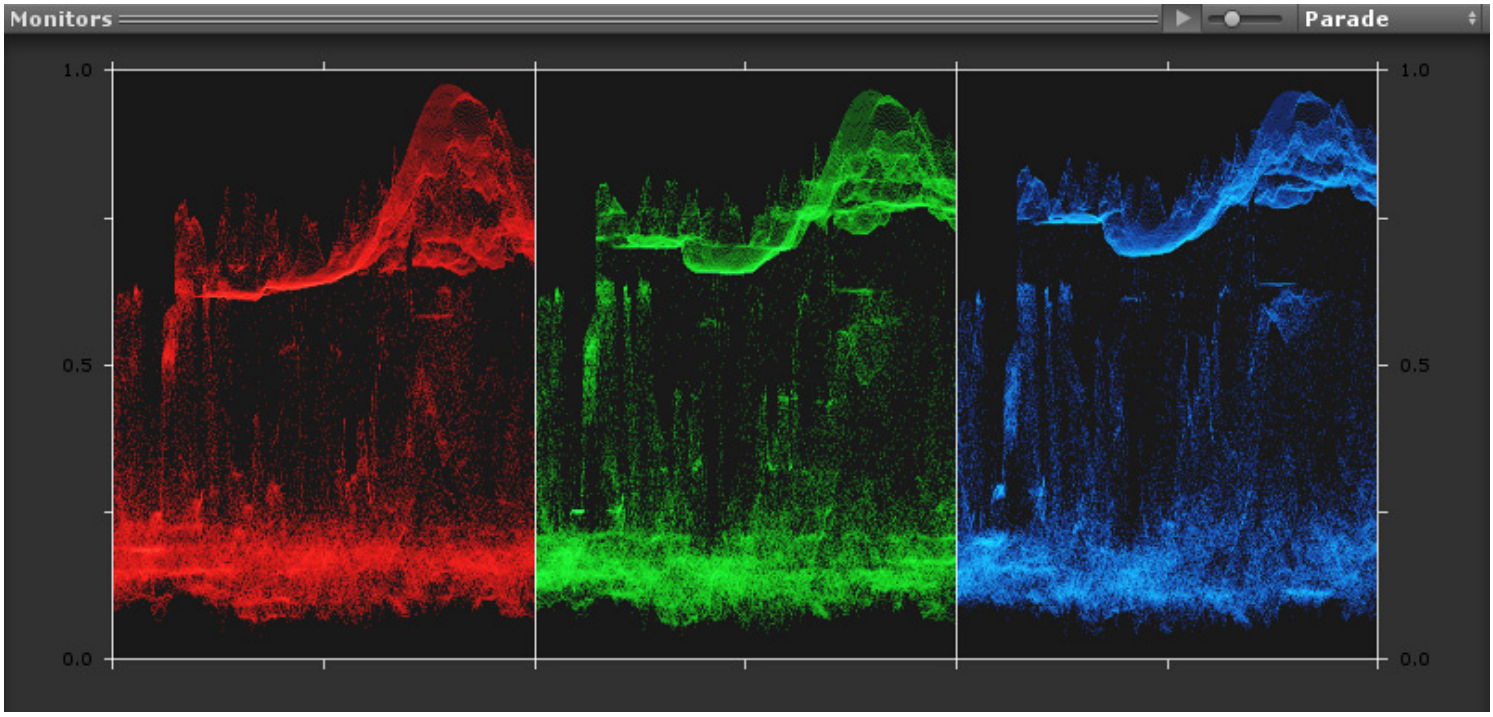


The Waveform Monitor

Parade

The Parade monitor is similar to the Waveform only it splits the image into red, green and blue separately.

It is useful in seeing the overall RGB balance in your image, for instance, if there is an obvious offset in one particular channel, and in making sure objects and elements in the shot that should be black, or white are true black or true white. Something that is true black, white (or grey for that matter) will have equal values across all channels.

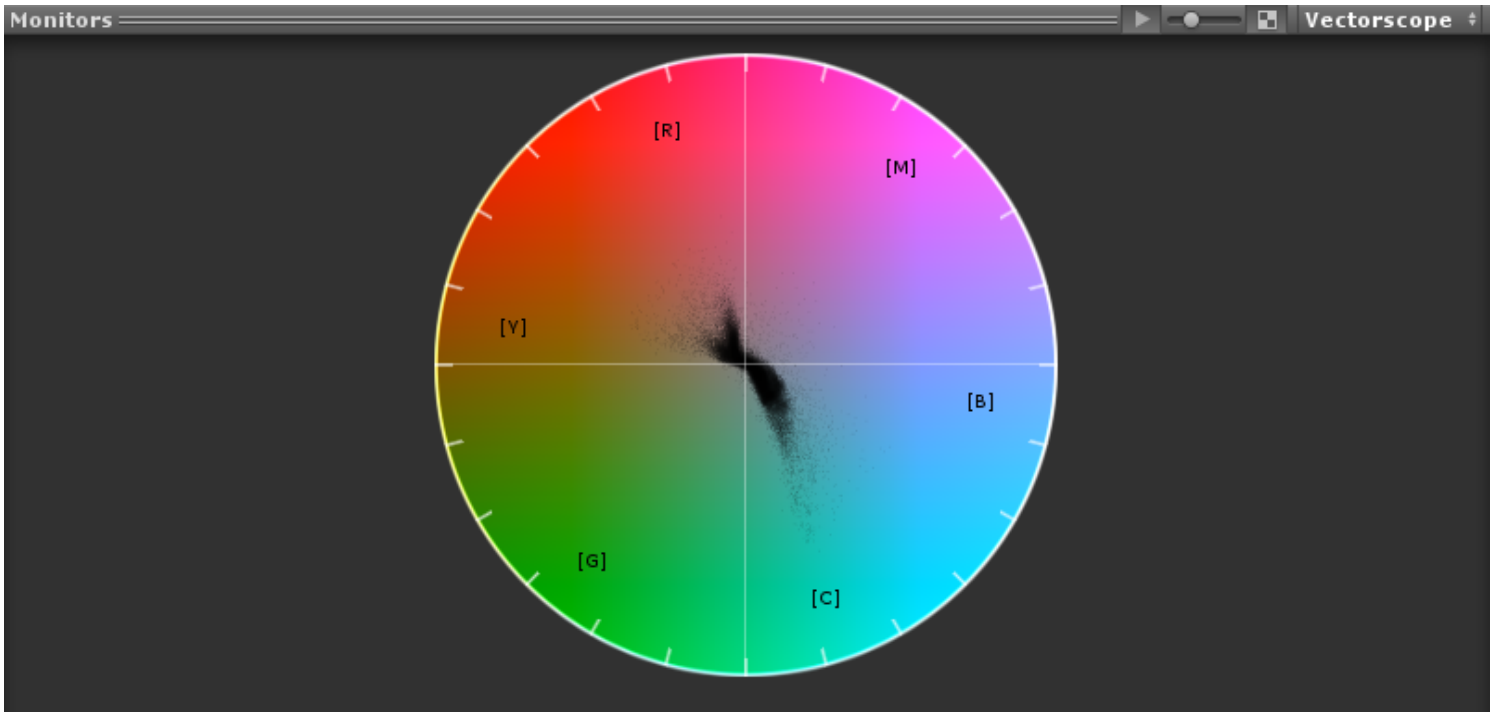


The Parade Monitor

Vectorscope

This monitor measures the overall range of hue (marked at yellow, red, magenta, blue, cyan and green) and saturation within the image. Measurements are relative to the center of the scope.

More saturated colors in the frame stretch those parts of the graph farther toward the edge, while less saturated colors remain closer to the center of the Vectorscope which represents absolute zero saturation. By judging how many parts of the Vectorscope graph stick out at different angles, you can see how many hues there are in the image. Furthermore, by judging how well centered the middle of the Vectorscope graph is relative to the absolute center, you can get an idea of whether there's a color imbalance in the image. If the Vectorscope graph is off-centered, the direction in which it leans lets you know that there's a color cast (tint) in the render.



The Vectorscope Monitor

Requirements

Compute shader

Shader model 3

See the Graphics Hardware Capabilities and Emulation page for further details and a list of compliant hardware.

2017–05–24 Page published with no editorial review

New feature in 5.6

Writing post-processing effects

Leave feedback

Post-processing is a way of applying effects to rendered images in Unity.

Any Unity script that uses the OnRenderImage function can act as a post-processing effect. Add it to a Camera GameObject for the script to perform post-processing.

OnRenderImage function

The OnRenderImage Unity Scripting API function receives two arguments:

The source image as a RenderTexture

The destination it should render into, which is a RenderTexture as well.

Post-processing effects often use Shaders. These read the source image, do some calculations on it, and render the result into the destination (using Graphics.Blit](../ScriptReference/Graphics.Blit.html), for example). The post-processing effect fully replaces all the pixels of the destination.

Cameras can have multiple post-processing effects, each as components. Unity executes them as a stack, in the order they are listed in the Inspector with the post-processing component at the top of the Inspector rendered first. In this situation, the result of the first post-processing component is passed as the "source image" to the next post-processing component. Internally, Unity creates one or more temporary **render textures** to keep these intermediate results in.

Note that the list of post-processing components in the post-processing stack do not specify the order they are applied in.

Things to keep in mind:

The destination render texture can be null, which means "render to screen" (that is, the back buffer). This happens on the last post-processing effect on a Camera.

When `OnRenderImage` finishes, Unity expects that the destination render texture is the active render target. Generally, a Graphics.Blit or manual **rendering** into the destination texture should be the last **rendering** operation.

Turn off depth buffer writes and tests in your post-processing effect shaders. This ensures that Graphics.Blit does not write unintended values into destination Z buffer. Almost all post-processing shader passes should contain `Cull Off ZWrite Off ZTest Always` states.

To use stencil or depth buffer values from the original **scene** render, explicitly bind the depth buffer from the original **scene** render as your depth target, using Graphics.SetRenderTarget. Pass the very first source image effects depth buffer as the depth buffer to bind.

After opaque post-processing effects

By default, Unity executes post-processing effects after it renders a whole Scene. In some cases, you may prefer Unity to render post-processing effects after it has rendered all opaque objects in your scene but before it renders others (for example, before skybox or transparencies). Depth-based effects like **Depth of Field** often use this.

To do this, add an ImageEffectOpaque attribute on the OnRenderImage Unity Scripting API function.

Texture coordinates on different platforms

If a post-processing effect is sampling different screen-related textures at once, you might need to be aware of how different platforms use texture coordinates. A common scenario is that the effect "source" texture and camera's depth texture need different vertical coordinates, depending on anti-aliasing settings. See the Unity User Manual Platform Differences page for more information.

Related topics

Depth Textures are often used in image post-processing to get distance to closest opaque surface for each **pixel** on screen.

For HDR rendering, a ImageEffectTransformsToLDR attribute indicates using **tonemapping**.

You can also use Command Buffers to perform post-processing.

Use RenderTexture.GetTemporary to get temporary render textures and do calculations inside a post-processing effect.

See also the Unity User Manual page on Writing Shader Programs.

2017–05–24 Page published with no editorial review

New feature in 5.6

High Dynamic Range Rendering

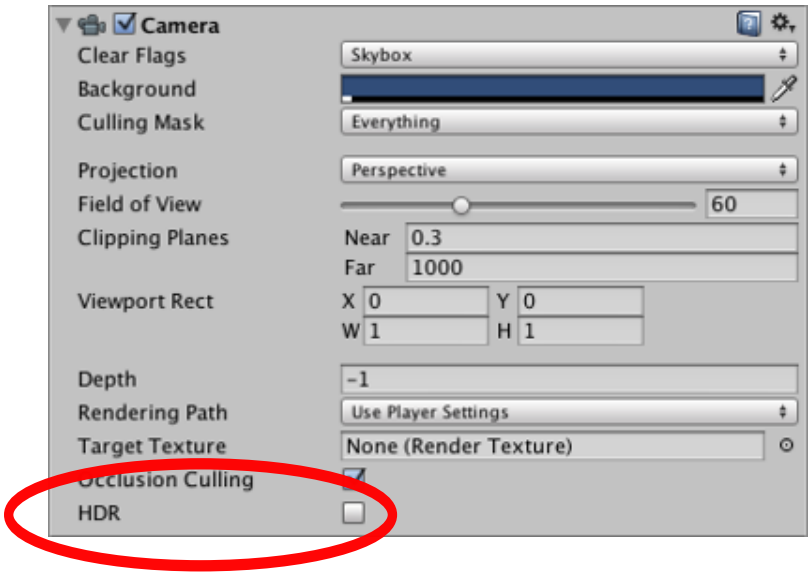
Leave feedback

In standard **rendering**, the red, green and blue values for a **pixel** are each represented by a fraction in the range 0..1, where 0 represents zero intensity and 1 represents the maximum intensity for the display device. While this is straightforward to use, it doesn't accurately reflect the way that lighting works in a real life **scene**. The human eye tends to adjust to local lighting conditions, so an object that looks white in a dimly lit room may in fact be less bright than an object that looks grey in full daylight. Additionally, the eye is more sensitive to brightness differences at the low end of the range than at the high end.

More convincing visual effects can be achieved if the rendering is adapted to let the ranges of pixel values more accurately reflect the light levels that would be present in a real scene. Although these values will ultimately need to be mapped back to the range available on the display device, any intermediate calculations (such as Unity's image effects) will give more authentic results. Allowing the internal representation of the graphics to use values outside the 0..1 range is the essence of **High Dynamic Range (HDR)** rendering.

Working with HDR

HDR is enabled separately for each **camera** using a setting on the **Camera component**:



When **HDR** is active, the scene is rendered into an **HDR** image buffer which can accommodate pixel values outside the 0..1 range. This buffer is then used by post-processing effects such as the Bloom effect in the Post-processing stack. The HDR image is then converted into the standard low dynamic range (LDR) image to be sent for display. This is usually done via **Tonemapping**, part of the Color Grading pipeline. The conversion to LDR must be applied at some point in the post-process pipeline but it need not be the final step if LDR-only post-processing effects are to be applied afterwards. For convenience, some post-processing effects can automatically convert to LDR after applying an HDR effect (see Scripting below).

Tonemapping

Tonemapping is the process of mapping HDR values back into the LDR range. There are many different techniques, and what is good for one project may not be the best for another. A variety of tonemapping techniques have been included in the Post-processing stack. To use them you can download the **Post-processing stack** from the Asset Store. A detailed description of the tonemapping types can be found in the Color Grading documentation.

An exceptionally bright scene rendered in HDR. Without tonemapping, most pixels seem out of range.

The same scene as above. But this time, tonemapping is bringing most intensities into a more plausible range. Note that adaptive tonemapping can even blend between above and this image thus simulating the adaptive nature of capturing media (e.g. eyes, cameras).

Advantages of HDR

Colors not being lost in high intensity areas
Better bloom and glow support
Reduction of banding in low frequency lighting areas

Disadvantages of HDR

Uses floating point **render textures** (rendering is slower and requires more VRAM)
No hardware anti-aliasing support (but you can use an Anti-Aliasing post-processing effect to smooth out the edges)
Not supported on all hardware

Usage notes

Forward Rendering

In **forward rendering** mode HDR is only supported if you have a post-processing effect present. This is due to performance considerations. If you have no post-processing effect present then no tonemapping will exist and intensity truncation will occur. In this situation the scene will be rendered directly to the backbuffer where HDR is not supported.

Deferred Rendering

In HDR mode the lighting buffer is also allocated as a floating point buffer. This reduces banding in the lighting buffer. HDR is supported in deferred rendering even if no post-processing effects are present.

Scripting

The **ImageEffectTransformsToLDR** attribute can be added to a post-processing effect script to indicate that the target buffer should be in LDR instead of HDR. Essentially, this means that a script can automatically convert to LDR after applying its HDR post-processing effect. See Writing Post-processing Effects for more details.

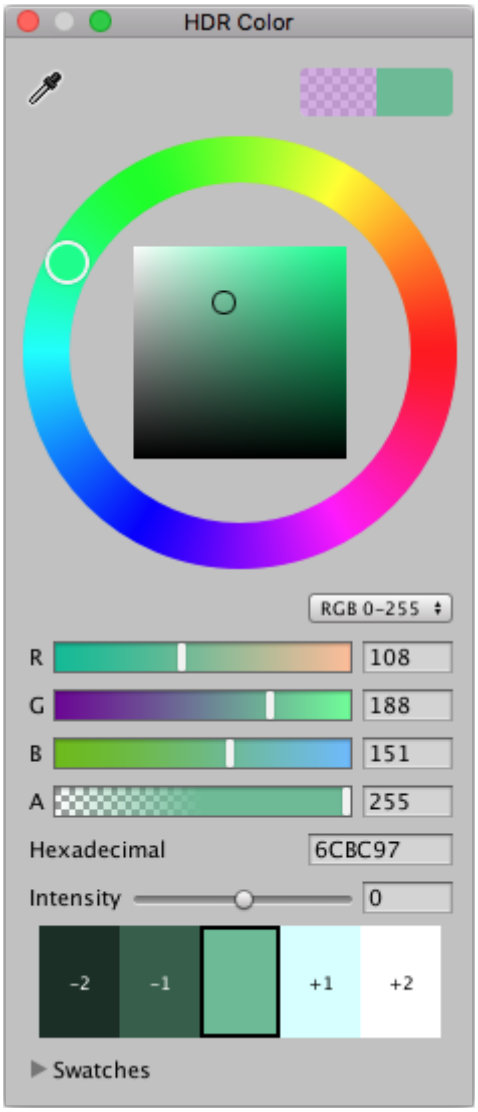
See also

HDR Color Picker.

HDR color picker

Leave feedback

The **HDR** Color picker looks similar to the ordinary Color picker, but it contains additional controls for adjusting the color's exposure.



Property:	Function:
Mode (default: **RGB 0–255**)	When using **HSV** or **RGB 0–255** mode, the color picker treats exposure adjustments independently from color channel data. However, the color channel data displayed in **RBG (0–1.0)** mode reflects the results of your exposure adjustment on the color data. Unlike the ordinary color picker, you can directly enter float values greater than 1.0 when editing color channels in **RGB 0–1.0** mode. In this case, the color picker derives the **Intensity** value automatically from the value you set.
RGBA	Use the slider or text box to define a RGBA value. The **Hexadecimal** value automatically updates to reflect the RGBA values.
Hexadecimal	Use the text box to define a hexadecimal value. The **RGBA** values automatically update to reflect the hexadecimal value.
Intensity	Use the **Intensity** slider to overexpose or underexpose the color. Each positive step along the slider provides twice as much light as the previous slider position, and each negative step provides half as much light.

Property:	Function:
Swatches	Use the exposure swatches under the **Intensity** slider to preview what the current color value looks like within a range of two steps in either direction. To quickly adjust the color's exposure, click a preview swatch.

Whenever you close the HDR Color window and reopen it, the window derives the color channel and intensity values from the color you are editing. Because of this, you might see slightly different values for the color channels in **HSV** and **RGB 0–255** mode or for the Intensity slider, even though the color channel values in **RGB 0–1.0** mode are the same as the last time you edited the color.

2018–07–27 Page amended with editorial review

HDR color picker updated in 2018.1

Rendering Paths

Leave feedback

Unity supports different ***Rendering** Paths*. You should choose which one you use depending on your game content and target platform / hardware. Different **rendering** paths have different performance characteristics that mostly affect Lights and Shadows. See render pipeline for technical details.

The **rendering path** used by your project is chosen in Graphics Settings. Additionally, you can override it for each Camera.

If the graphics card can't handle a selected rendering path, Unity will automatically use a lower fidelity one. For example, on a GPU that can't handle **Deferred Shading**, **Forward Rendering** will be used.

Deferred Shading

Deferred Shading is the rendering path with the most lighting and shadow fidelity, and is best suited if you have many realtime lights. It requires a certain level of hardware support.

For more details see the Deferred Shading page.

Forward Rendering

Forward is the traditional rendering path. It supports all the typical Unity graphics features (normal maps, per-pixel lights, shadows etc.). However under default settings, only a small number of the brightest lights are rendered in per-pixel lighting mode. The rest of the lights are calculated at object vertices or per-object.

For more details see the Forward Rendering page.

Legacy Deferred

Legacy Deferred (light prepass) is similar to Deferred Shading, just using a different technique with different trade-offs. It does not support the Unity 5 physically based **standard shader**.

For more details see the Deferred Lighting page.

Legacy Vertex Lit

Legacy Vertex Lit is the rendering path with the lowest lighting fidelity and no support for realtime shadows. It is a subset of Forward rendering path.

For more details see the Vertex Lit page.

NOTE: *Deferred rendering is not supported when using Orthographic projection. If the camera's projection mode is set to Orthographic, these values are overridden, and the camera will always use Forward rendering.*

Rendering Paths Comparison

	Deferred	*Forward*	**Legacy Deferred**	*Vertex Lit*
Features				

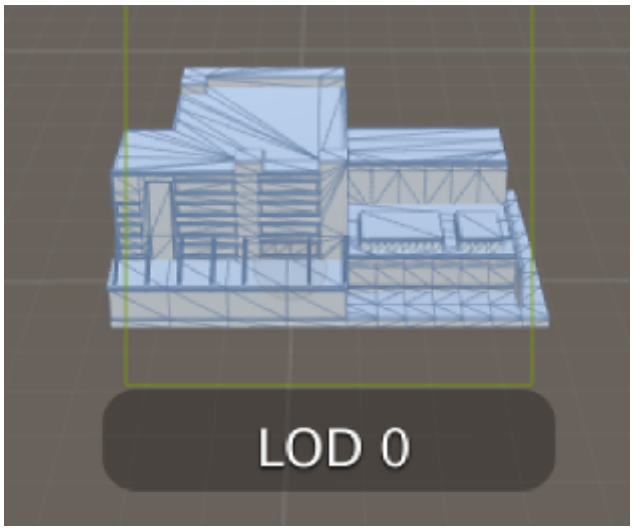
	Deferred	*Forward*	**Legacy Deferred**	*Vertex Lit*
Per-pixel lighting (normal maps, light cookies)	Yes	Yes	Yes	-
Realtime shadows	Yes	With caveats	Yes	-
Reflection Probes	Yes	Yes	-	-
Depth&Normals Buffers	Yes	Additional render passes	Yes	-
Soft Particles	Yes	-	Yes	-
Semitransparent objects	-	Yes	-	Yes
Anti-Aliasing	-	Yes	-	Yes
Light **Culling Masks**	Limited	Yes	Limited	Yes
Lighting Fidelity	All per-pixel	Some per-pixel	All per-pixel	All per-vertex
Performance				
Cost of a per-pixel Light	Number of **pixels** it illuminates	Number of **pixels** * Number of objects it illuminates	Number of **pixels** it illuminates	-
Number of times objects are normally rendered	1	Number of per-pixel lights	2	1
Overhead for simple **scenes**	High	None	Medium	None
Platform Support				
PC (Windows/Mac)	**Shader** Model 3.0+ & MRT	All	**Shader** Model 3.0+	All
Mobile (iOS/Android)	OpenGL ES 3.0 & MRT, Metal (on devices with A8 or later SoC)	All	OpenGL ES 2.0	All
Consoles	XB1, **PS4**	All	XB1, **PS4**, 360	-

Level of Detail (LOD)

Leave feedback

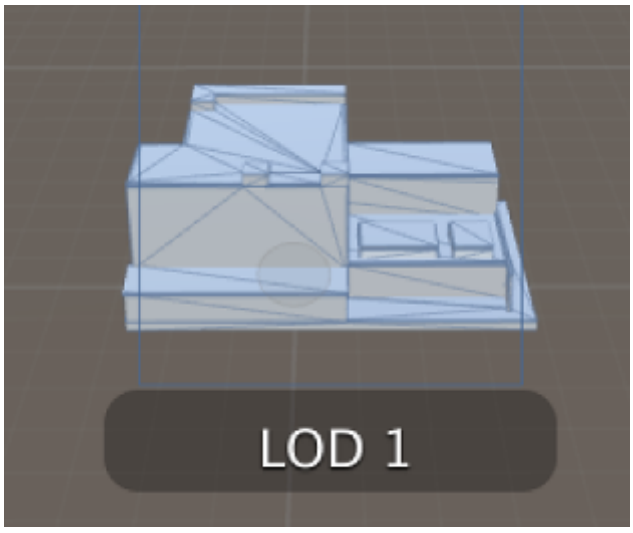
When a **GameObject** in the **scene** is a long way from the **camera**, the amount of detail that can be seen on it is greatly reduced. However, the same number of triangles will be used to render the object, even though the detail will not be noticed. An optimisation technique called *Level Of Detail* (LOD) **rendering** allows you to reduce the number of triangles rendered for an object as its distance from the camera increases. As long as your objects aren't all close to the camera at the same time, **LOD** will reduce the load on the hardware and improve **rendering** performance.

In Unity, you use the **LOD Group component** to set up LOD rendering for an object. Full details are given on the component reference page but the images below show how the LOD level used to render an object changes with its distance from camera. The first shows LOD level 0 (the most detailed). Note the large number of small triangles in the **mesh**:



camera at LOD 0

The second shows a lower level being used when the object is farther away. Note that the mesh has been reduced in detail (smaller number of larger triangles):



camera at LOD 1

Since the arrangement of LOD levels depends somewhat on the target platform and available rendering performance, Unity lets you set maximum LOD levels and a LOD bias preference (ie, whether to favour higher or lower LOD levels at threshold distances) in the Quality Settings.

LOD Naming Convention for Importing Objects

If you create a set of meshes with names ending in _LOD0, _LOD1, _LOD2, **etc**, for as many LOD levels as you like, a LOD group for the object with appropriate settings will be created for you automatically on import. For example, if the base name for your mesh is *Player*, you could create files called *Player_LOD0*, *Player_LOD1* and *Player_LOD2* to generate an object with three LOD levels. The numbering convention assumes that LOD 0 is the most detailed model and increasing numbers correspond to decreasing detail.

Graphics API support

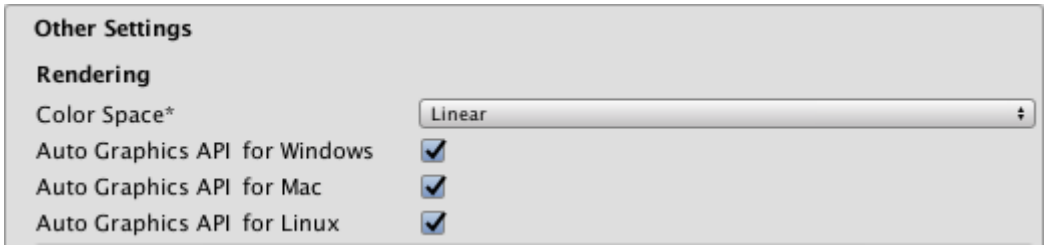
Leave feedback

Unity supports DirectX, Metal, OpenGL, and Vulkan graphics APIs, depending on the availability of the API on a particular platform. Unity uses a built-in set of graphics APIs, or the graphics APIs that you select in the Editor.

To use Unity's default graphics APIs:

Open the Player Settings (menu: **Edit > Project Settings > Player**).

Navigate to **Other Settings** and make sure **Auto Graphics API** is checked:

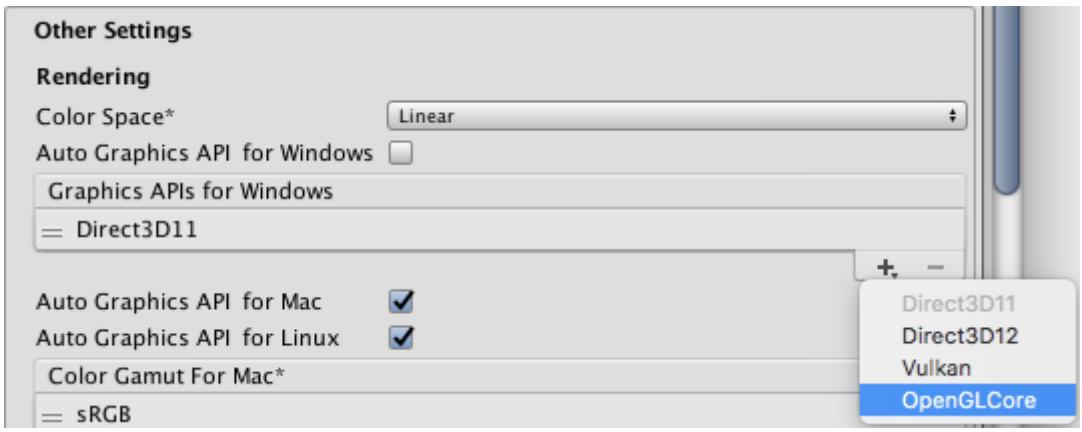


Using the default graphics APIs

When **Auto Graphics API** for a platform is checked, the Player build includes a set of built-in graphics APIs and uses the appropriate one at run time to produce a best case scenario.

When **Auto Graphics API** for a platform is not checked, the Editor uses the first API in the list. So, for example, to see how your application runs on OpenGL in the Editor, move **OpenGLCore** to the top of the list and the Editor switches to use OpenGL **rendering**.

To override the default graphics APIs and use an alternate graphics API for the Editor and Player, uncheck the relevant **Auto Graphics API**, click the plus (+) button and choose the graphics API from the drop-down menu.



Adding OpenGLCore to the Graphics APIs for Windows list

The graphics API at the top of the **Auto Graphics API** list is the default API. If the default API isn't supported by the specific platform, Unity uses the next API in the **Auto Graphics API** list.

For information on how graphics rendering behaves between the platforms and **Shader** language semantics, see Platform-specific rendering differences. Tessellation and geometry shaders are only supported by a subset of graphics APIs. This is controlled by the Shader Compilation Target level.

For graphics API specific information, see documentation on Metal, DirectX and OpenGL.

2018–06–02 Page published with editorial review

DirectX

Leave feedback

To set DirectX11 as your default Graphics API in the Editor or Standalone Player, go to the Player Settings (menu: **Edit > Project Settings > Player**) and navigate to **Other Settings**. Uncheck **Auto Graphics API for Windows**, and choose **DirectX11** from the list. For more details, see [Graphics API support](Graphics APIs).

Surface Shaders

Some parts of the Surface Shader compilation pipeline do not understand DX11-specific HLSL syntax, so if you're using HLSL features like StructuredBuffers, RWTextures and other non-DX9 syntax, you need to wrap it into a DX11-only preprocessor macro. See documentation on Platform-specific differences for more information.

Tessellation & Geometry Shaders

Surface **Shaders** have support for simple tessellation and displacement. See documentation on Surface Shader Tessellation for more information.

When manually writing Shader programs, you can use the full set of DX11 Shader model 5.0 features, including Geometry, Hull and Domain Shaders.

Tessellation and geometry shaders are only supported by a subset of graphics APIs. This is controlled by the Shader Compilation Target level.

Compute Shaders

Compute Shaders run on the graphics card and can speed up **rendering**. See documentation on Compute Shaders for more information.

2018–06–02 Page amended with editorial review

Metal

Leave feedback

Metal is the standard graphics API for Apple devices. Unity supports Metal on **iOS**, tvOS and macOS (Standalone and Editor).

Metal has a larger feature set on Apple platforms than OpenGL ES. See the advantages and disadvantages of using Metal below.

Advantages of using Metal

Lower CPU overhead of graphics API calls
API level validation layer
Better GPU control on multi-GPU systems
Supports memory-less render targets (on iOS/tvOS)
New Apple standard for Apple
Computer shaders
Tessellation shaders

Disadvantages of using Metal

No support for low-end devices
No support for geometry shaders

Limitations and requirements

iOS and tvOS have Metal support for Apple A7 or newer SoC-s.

macOS has Metal support for Intel HD and Iris Graphics from the HD 4000 series or newer, AMD GCN-based GPUs, and Nvidia Kepler-based GPUs or newer.

Minimum shader compilation target is 3.5.

Metal does not support geometry **shaders**.

Enabling Metal

To make the Unity Editor and Standalone Player use Metal as the default graphics API, do one of the following:

In the Editor, go to menu: **Edit** > **Project Settings** > **Player** and enable **Metal Editor Support**.

Or, if you are using MacOS, open **Terminal** and use the `-force-gfx-metal` command line argument.

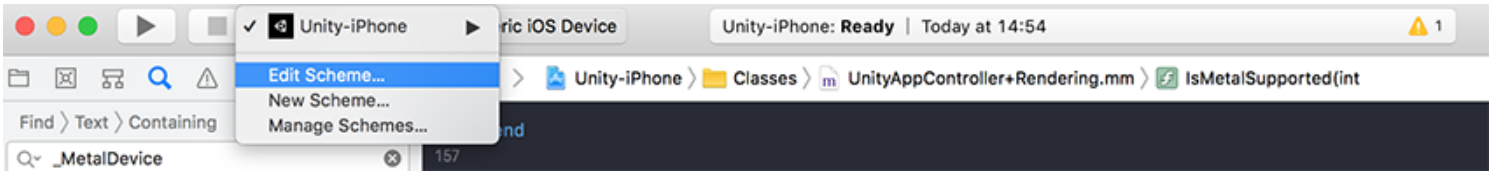
Metal is enabled by default on iOS, tvOS and macOS Standalone Players.

Validating Metal API

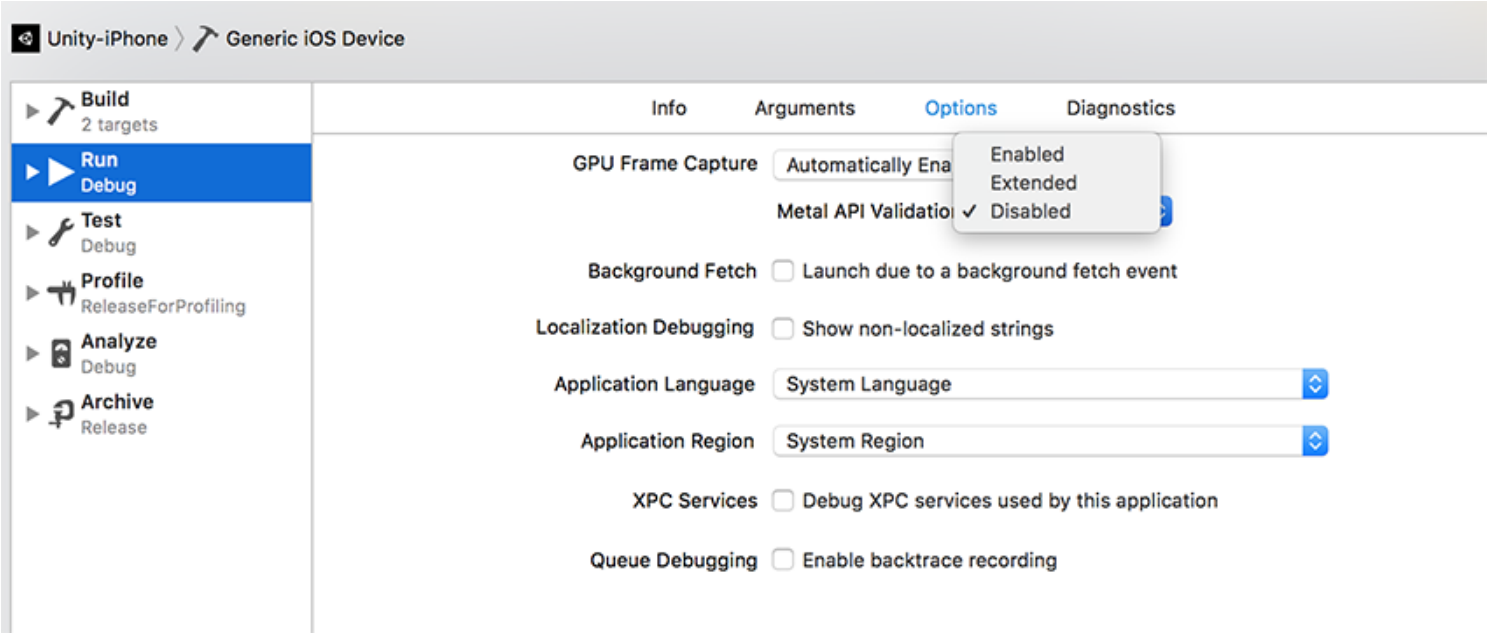
Xcode offers Metal API validation, which you can use to trace obscure issues. To enable Metal API validation in Xcode:

In Unity, build your Project for iOS. This generates an Xcode project.

Open the generated Xcode project in Xcode and select **Edit Scheme**.



Select **Run** > **Options** > **Metal API Validation** and choose **Enabled**



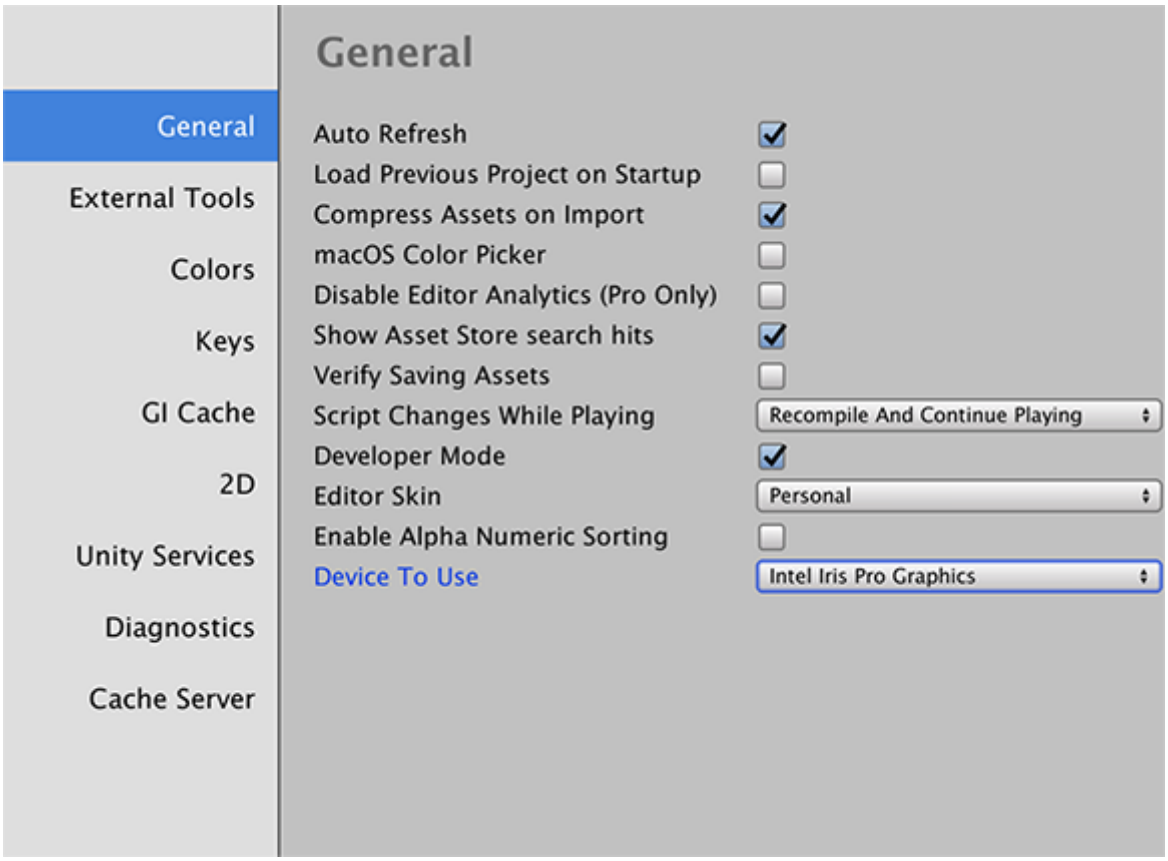
Validation errors break code execution in the XCode editor, and appear in device logs.

Note: Enabling validation increases CPU usage, so only enable it for debugging.

Selecting a GPU device

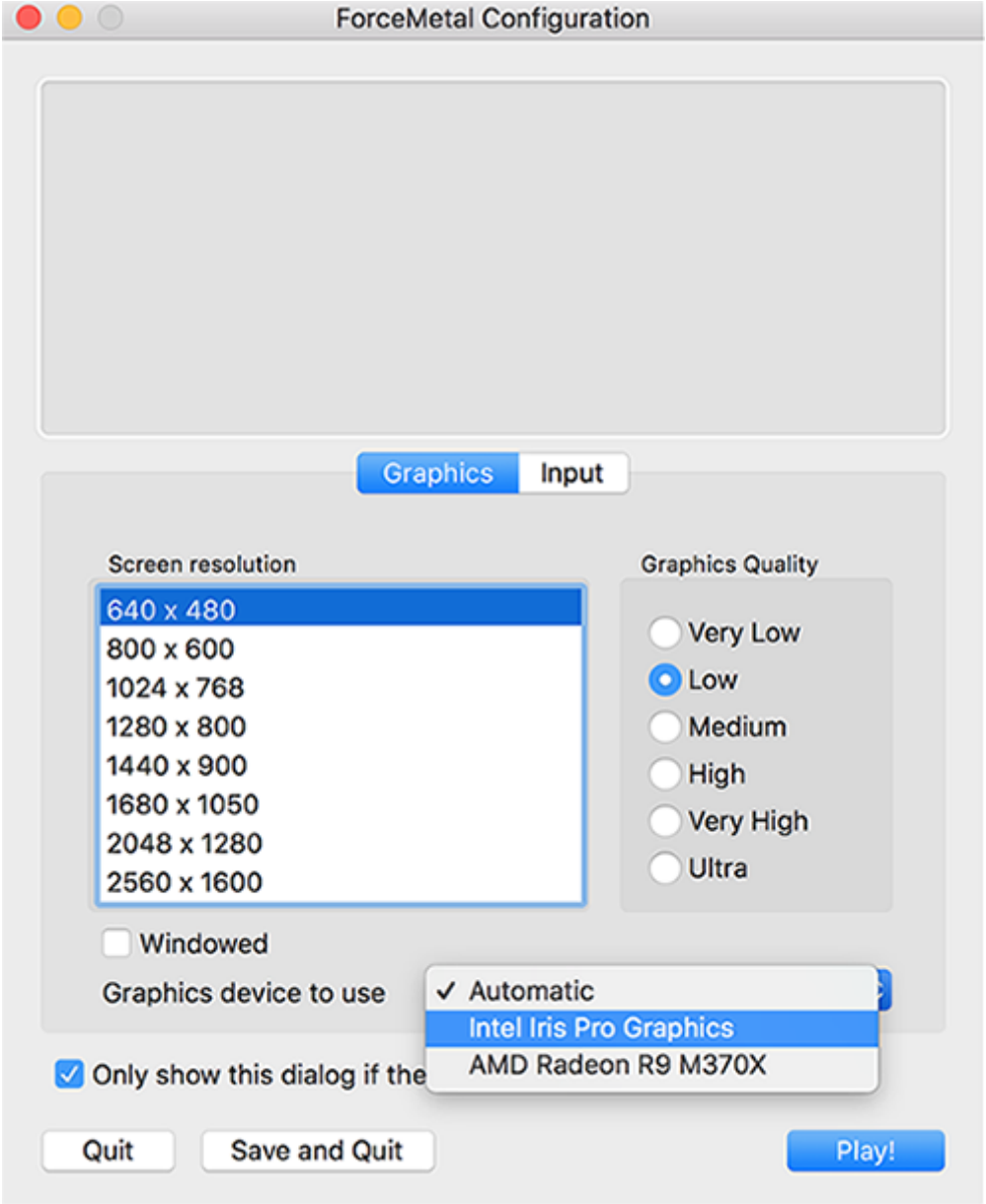
Metal allows you to select a GPU device when you run your application. This enables you to test your Project on different GPU setups, or save power by using a low power GPU.

To change the Unity Editor target GPU device, select menu: **Unity** > **Preferences...** > **General** and set the **Device To Use**:



Changing target GPU in the Editor

To change the Standalone Player target GPU device, start your application (or select menu: **File** > **Build and run**) and set the **Graphics device to use** to the relevant GPU in the dialog that appears:



Changing target GPU on Standalone Player

Using memoryless render targets

Metal allows you to use memory-less render targets to optimize memory on mobile devices introduced in iOS and tvOS 10.0. This enables you to render to a RenderTexture without backing it up in system memory, so contents are only temporarily stored in the on-tile memory during **rendering**.

For more information, see RenderTexture.memorylessMode.

2018–05–22 Page published with editorial review

Added advice on using Metal in 2017.4

OpenGL Core

Leave feedback

OpenGL Core is a back-end capable of supporting the latest OpenGL features on Windows, MacOS X and Linux. This scales from OpenGL 3.2 to OpenGL 4.5, depending on the OpenGL driver support.

Enabling OpenGL Core

To set OpenGL Core as your default Graphics API in the Editor or Standalone Player, go to the Player Settings (menu: **Edit > Project Settings > Player**), and navigate to **Other Settings**. Uncheck **Auto Graphics API for Windows**, and choose **OpenGLCore** from the list. For more details, see [Graphics API support](Graphics APIs).

OpenGL requirements

OpenGL Core has the following minimum requirements:

Mac OS X 10.8 (OpenGL 3.2), MacOSX 10.9 (OpenGL 3.2 to 4.1)

Windows with NVIDIA since 2006 (GeForce 8), AMD since 2006 (Radeon HD 2000), Intel since 2012 (HD 4000 / IvyBridge) (OpenGL 3.2 to OpenGL 4.5)

Linux (OpenGL 3.2 to OpenGL 4.5)

macOS OpenGL driver limitations

The macOS OpenGL backend for the Editor and Standalone supports OpenGL 3.x and 4.x features such as tessellation and geometry **shaders**.

However, as Apple restricts the OpenGL version on OS X desktop to 4.1 at most, it does not support all DirectX 11 features (such as Unordered Access Views or Compute Shaders). This means that all shaders that are configured to target Shader Level 5.0 (with #pragma target 50) will fail to load on OS X.

Therefore a new shader target level is introduced: #pragma target gl4.1. This target level requires at least OpenGL 4.1 or DirectX 11.0 Shader Level 5 on desktop, or OpenGL ES 3.1 + Android Extension Pack on mobiles.

OpenGL Core features

The new OpenGL back-end introduces many new features (previously mostly DX11/GLES3 only):

Compute shaders (as well as ComputeBuffers and “random write” render textures)
Tessellation and Geometry shaders
Indirect draw (Graphics.DrawProcedural and Graphics.DrawProceduralIndirect)
Advanced blend modes

Shader changes

When using the existing #pragma targets, they map to following GL levels:

#pragma target 4.0 // OpenGL ES 3.1, desktop OpenGL 3.x, DX Shader Model 4.0
#pragma target gl4.1 // Desktop OpenGL 4.1, SM 4.0 + tessellation to match MacOSX 10.9 capabilities

#pragma target 5.0 // OpenGL ES 3.1 + Android Extension Pack, desktop OpenGL >= 4.2, DX Shader Model 5.0

For including and excluding shader platforms from using a specific shaders, the following #pragma only_renderers / exclude_renderers targets can be used:

#pragma only_renderers glcore: Only compile for the desktop GL. Like the ES 3 target, this also scales up to contain all desktop GL versions, where basic shaders will support GL 2.x while shaders requiring SM5.0 features require OpenGL 4.2+.

OpenGL core profile command line arguments

It's possible to start the editor or the player with OpenGL using the command line arguments:

-force-opengl: To use the legacy OpenGL back-end
-force-glcore: To use the new OpenGL back-end. With this argument, Unity will detect all the features the platform support to run with the best OpenGL version possible and all available OpenGL extensions
-force-glcoreXY: XY can be 32, 33, 40, 41, 42, 43, 44 or 45; each number representing a specific version of OpenGL. If the platform doesn't support a specific version of OpenGL, Unity will fallback to a supported version
-force-clamped: Request that Unity doesn't use OpenGL extensions which guarantees that multiple platforms will execute the same code path. This is an approach to test if an issue is platform specific (a driver bug for example).

Native OpenGL ES on desktop command line arguments

OpenGL ES graphics API is available on Windows machines with Intel or NVIDIA GPUs with drivers supporting OpenGL ES.

-force-gles: To use the new OpenGL back-end in OpenGL ES mode. With this argument, Unity will detect all the features the platform support to run with the best OpenGL ES version possible and all available OpenGL ES extensions
-force-glesXY: XY can be 20, 30, 31 or 31aep; each number representing a specific version of OpenGL ES. If the platform doesn't support a specific version of OpenGL ES, Unity will fallback to a supported version. If the platform doesn't support OpenGL ES, Unity will fallback to another graphics API.
-force-clamped: Request that Unity doesn't use OpenGL extensions which guarantees that multiple platforms will execute the same code path. This is an approach to test if an issue is platform specific (a driver bug for example).

2018–06–02 Page amended with editorial review

Compute shaders

Leave feedback

SWITCH TO SCRIPTING

Compute **shaders** are programs that run on the graphics card, outside of the normal **rendering** pipeline. They can be used for massively parallel GPGPU algorithms, or to accelerate parts of game **rendering**. In order to efficiently use them, an in-depth knowledge of GPU architectures and parallel algorithms is often needed; as well as knowledge of DirectCompute, OpenGL Compute, CUDA, or OpenCL.

Compute shaders in Unity closely match DirectX 11 DirectCompute technology. Platforms where compute shaders work:

Windows and Windows Store, with a DirectX 11 or DirectX 12 graphics API and Shader Model 5.0 GPU

macOS and **iOS** using Metal graphics API

Android, Linux and Windows platforms with Vulkan API

Modern OpenGL platforms (OpenGL 4.3 on Linux or Windows; OpenGL ES 3.1 on Android). Note that Mac OS X does not support OpenGL 4.3

Modern consoles (Sony PS4 and Microsoft Xbox One)

Compute shader support can be queried runtime using SystemInfo.supportsComputeShaders.

Compute shader Assets

Similar to regular shaders, compute shaders are **Asset** files in your project, with a *.compute* file extension. They are written in DirectX 11 style HLSL language, with a minimal number of #pragma compilation directives to indicate which functions to compile as compute shader kernels.

Here's a basic example of a compute shader file, which fills the output texture with red:

```
// test.compute

#pragma kernel FillWithRed

RWTexture2D<float4> res;

[numthreads(1,1,1)]
void FillWithRed (uint3 dtid : SV_DispatchThreadID)
{
    res[dtid.xy] = float4(1,0,0,1);
}
```

The language is standard DX11 HLSL, with an additional `#pragma kernel FillWithRed` directive. One compute shader Asset file must contain at least one`compute kernel` that can be invoked, and that function is indicated by

the `#pragma directive`. There can be more kernels in the file; just add multiple `#pragma kernel` lines.

When using multiple `#pragma kernel` lines, note that comments of the style `// text` are not permitted on the same line as the `#pragma kernel` directives, and cause compilation errors if used.

The `#pragma kernel` line can optionally be followed by a number of preprocessor macros to define while compiling that kernel, for example:

```
#pragma kernel KernelOne SOME_DEFINE DEFINE_WITH_VALUE=1337
#pragma kernel KernelTwo OTHER_DEFINE
// ...
```

Invoking compute shaders

In your script, define a variable of ComputeShader type and assign a reference to the Asset. This allows you to invoke them with ComputeShader.Dispatch function. See Unity documentation on ComputeShader class for more details.

Closely related to compute shaders is a ComputeBuffer class, which defines arbitrary data buffer ("structured buffer" in DX11 lingo). Render Textures can also be written into from compute shaders, if they have "random access" flag set ("unordered access view" in DX11). See RenderTexture.enableRandomWrite to learn more about this.

Texture samplers in compute shaders

Textures and samplers aren't separate objects in Unity, so to use them in compute shaders you must follow one of the following Unity-specific rules:

Use the same name as the Texture name, with `sampler` at the beginning (for example, `Texture2D MyTex`; `SamplerState samplerMyTex`). In this case, the sampler is initialized to that Texture's filter/wrap/aniso settings.

Use a predefined sampler. For this, the name has to have `Linear` or `Point` (for filter mode) and `Clamp` or `Repeat` (for wrap mode). For example, `SamplerState MyLinearClampSampler` creates a sampler that has Linear filter mode and Clamp wrap mode.

For more information, see documentation on Sampler States.

Cross-platform support

As with regular shaders, Unity is capable of translating compute shaders from HLSL to other shader languages. Therefore, for the easiest cross-platform builds, you should write compute shaders in HLSL. However, there are some factors that need to be considered when doing this.

Cross-platform best practices

DirectX 11 (DX11) supports many actions that are not supported on other platforms (such as Metal or OpenGL ES). Therefore, you should always ensure your shader has well-defined behavior on platforms that offer less support, rather than only on DX11. Here are few things to consider:

Out-of-bounds memory accesses are bad. DX11 might consistently return zero when reading, and read some writes without issues, but platforms that offer less support might crash the GPU when doing this. Watch out for DX11-specific hacks, buffer sizes not matching with multiple of your thread group size, trying to read neighboring data elements from the beginning or end of the buffer, and similar incompatibilities.

Initialize your resources. The contents of new buffers and Textures are undefined. Some platforms might provide all zeroes, but on others, there could be anything including NaNs.

Bind all the resources your compute shader declares. Even if you know for sure that the shader does not use resources in its current state because of branching, you must still ensure a resource is bound to it.

Platform-specific differences

Metal (for iOS and tvOS platforms) does not support atomic operations on Textures. Metal also does not support `GetDimensions` queries on buffers. Pass the buffer size info as constant to the shader if needed.

OpenGL ES 3.1 (for (Android, iOS, tvOS platforms) only guarantees support for 4 compute buffers at a time. Actual implementations typically support more, but in general if developing for OpenGL ES, you should consider grouping related data in structs rather than having each data item in its own buffer.

HLSL-only or GLSL-only compute shaders

Usually, compute shader files are written in HLSL, and compiled or translated into all necessary platforms automatically. However, it is possible to either prevent translation to other languages (that is, only keep HLSL platforms), or to write GLSL compute code manually.

The following information only applies to HLSL-only or GLSL-only compute shaders, not cross-platform builds. This is because this information can result in compute shader source being excluded from some platforms.

Compute shader source surrounded by `CGPROGRAM` and `ENDCG` keywords is not processed for non-HLSL platforms.

Compute shader source surrounded by `GLSLPROGRAM` and `ENDGLSL` keywords is treated as GLSL source, and emitted verbatim. This only works when targeting OpenGL or GLSL platforms. You should also note that while automatically translated shaders follow HLSL data layout on buffers, manually written GLSL shaders follow GLSL layout rules.

2017–05–18 Page amended with limited editorial review

Added in 5.6: SystemInfo.supportsComputeShaders, platforms macOS, iOS (using Metal), Android, Linux, Windows (with Vulkan)

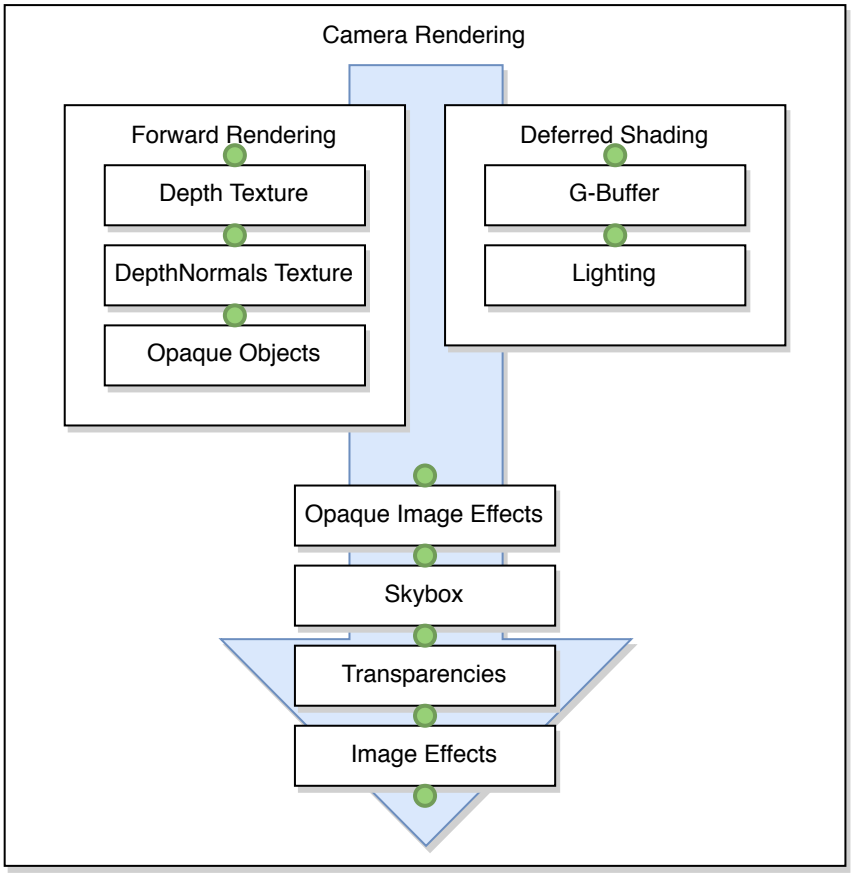
Graphics Command Buffers

Leave feedback

It is possible to extend Unity's rendering pipeline with so called "command buffers". A command buffer holds list of **rendering** commands ("set render target, draw mesh, ..."), and can be set to execute at various points during **camera** rendering.

For example, you could render some additional objects into deferred shading G-buffer after all regular objects are done.

A high-level overview of how cameras render **scene** in Unity is shown below. At each point marked with a green dot, you can add command buffers to execute your commands.



See CommandBuffer scripting class and CameraEvent enum for more details.

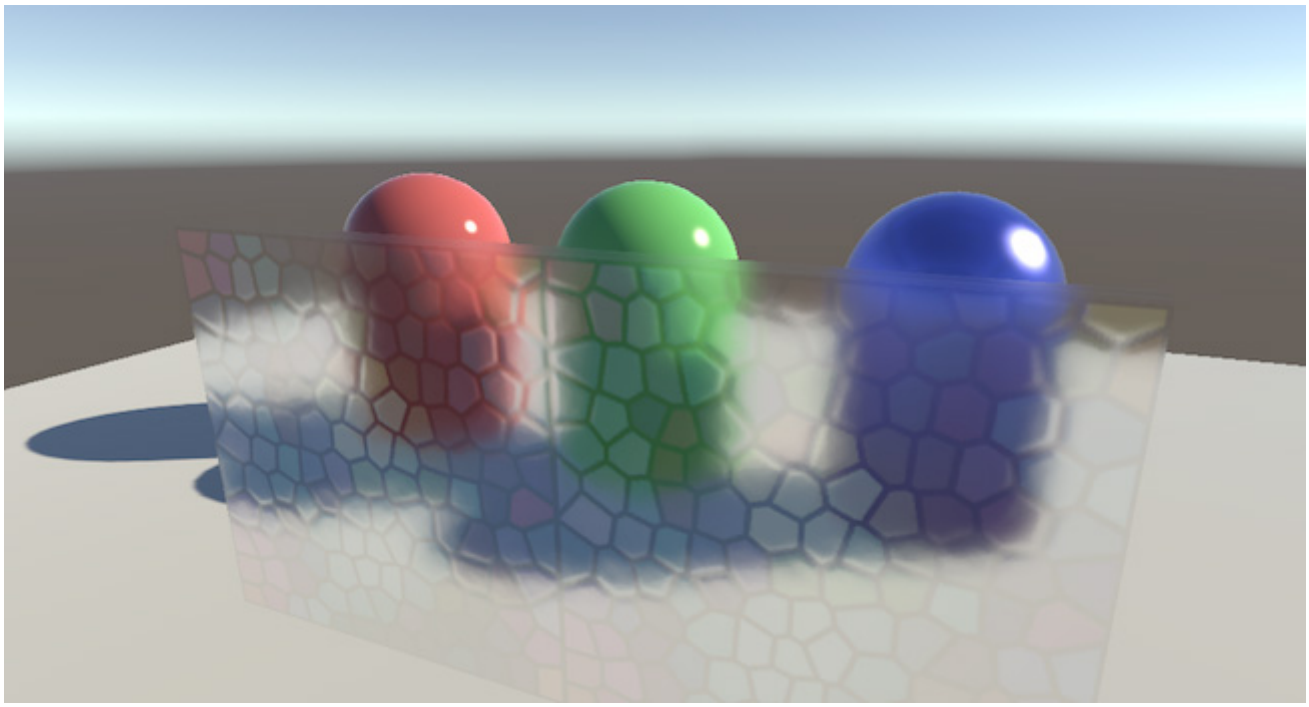
Command buffers can also be used as a replacement for, or in conjunction with image effects.

Example Code

Sample project demonstrating some of the techniques possible with command buffers: **RenderingCommandBuffers.zip** .

Blurry Refractions

This scene shows a "blurry refraction" technique.

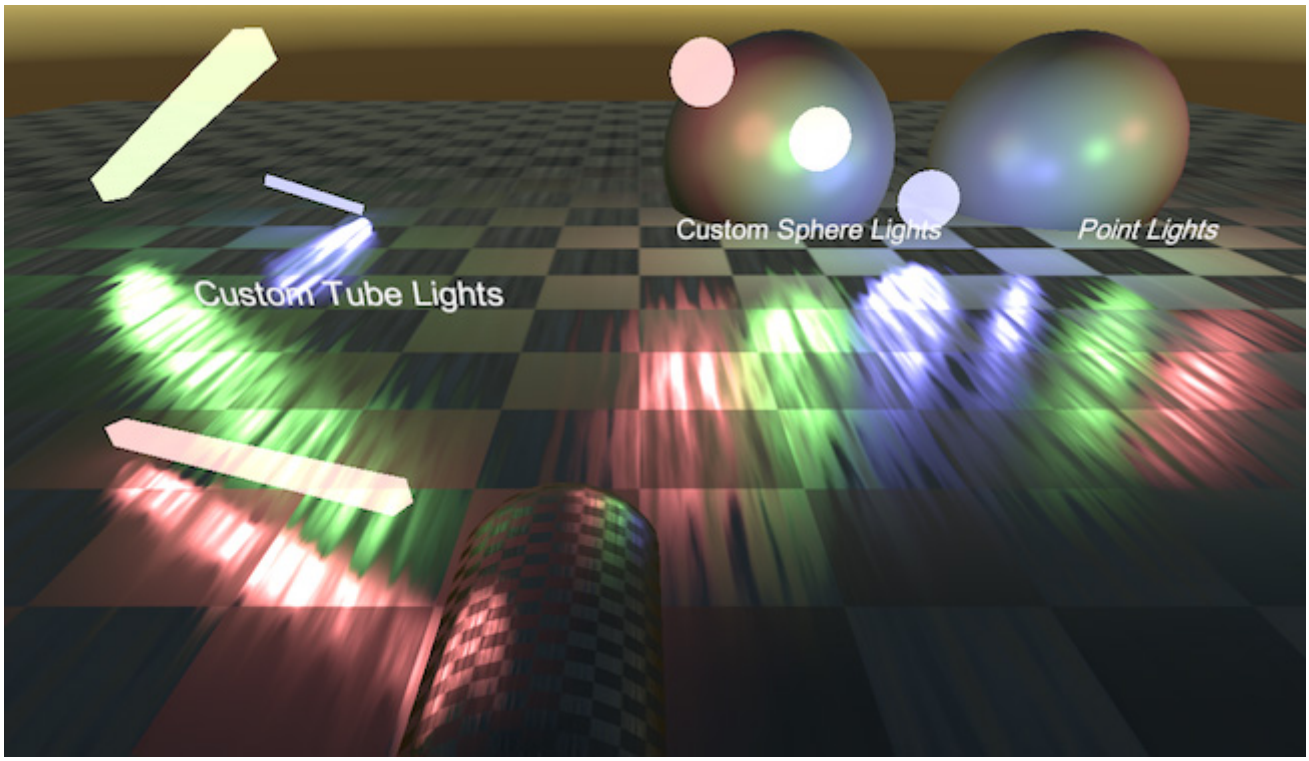


After opaque objects and **skybox** is rendered, current image is copied into a temporary render target, blurred and set up a global **shader** property. **Shader** on the glass object then samples that blurred image, with UV coordinates offset based on a normal map to simulate refraction.

This is similar to what shader GrabPass does does, except you can do more custom things (in this case, blurring).

Custom Area Lights in Deferred Shading

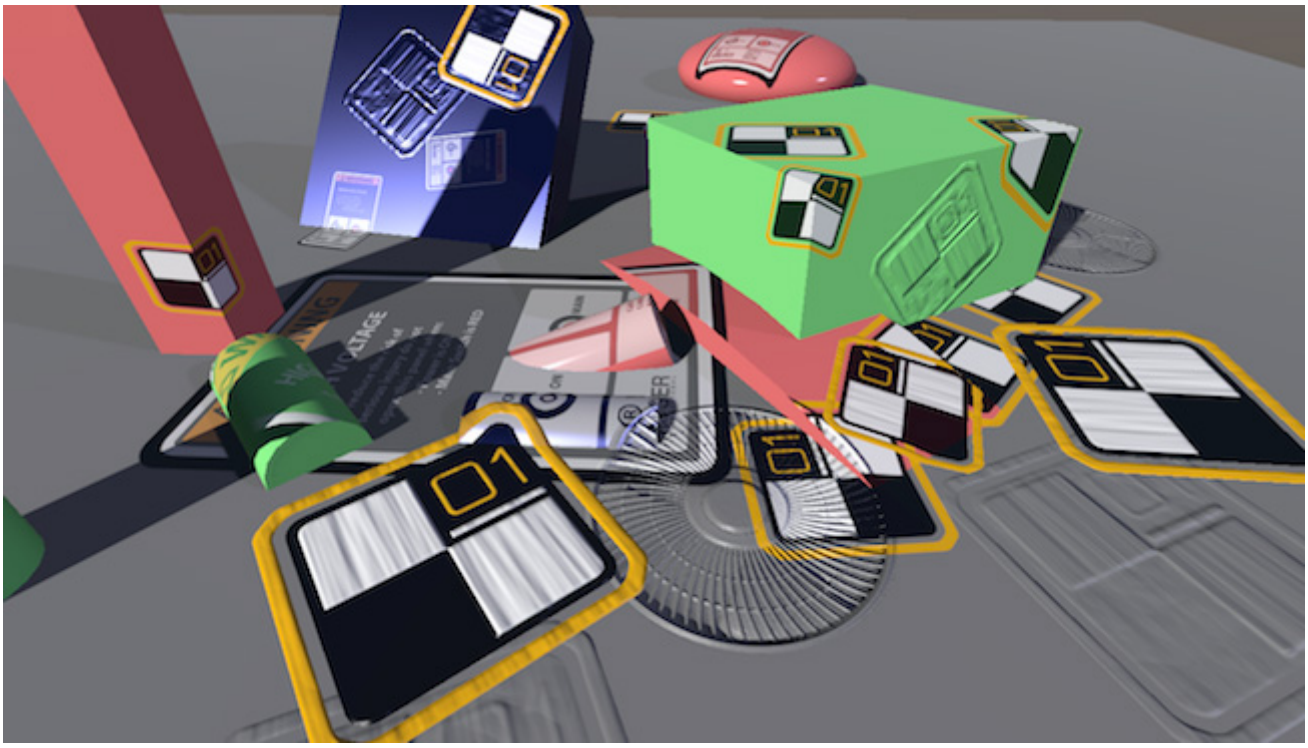
This scene shows an implementation of “custom deferred lights”: sphere-shaped lights, and tube-shaped lights.



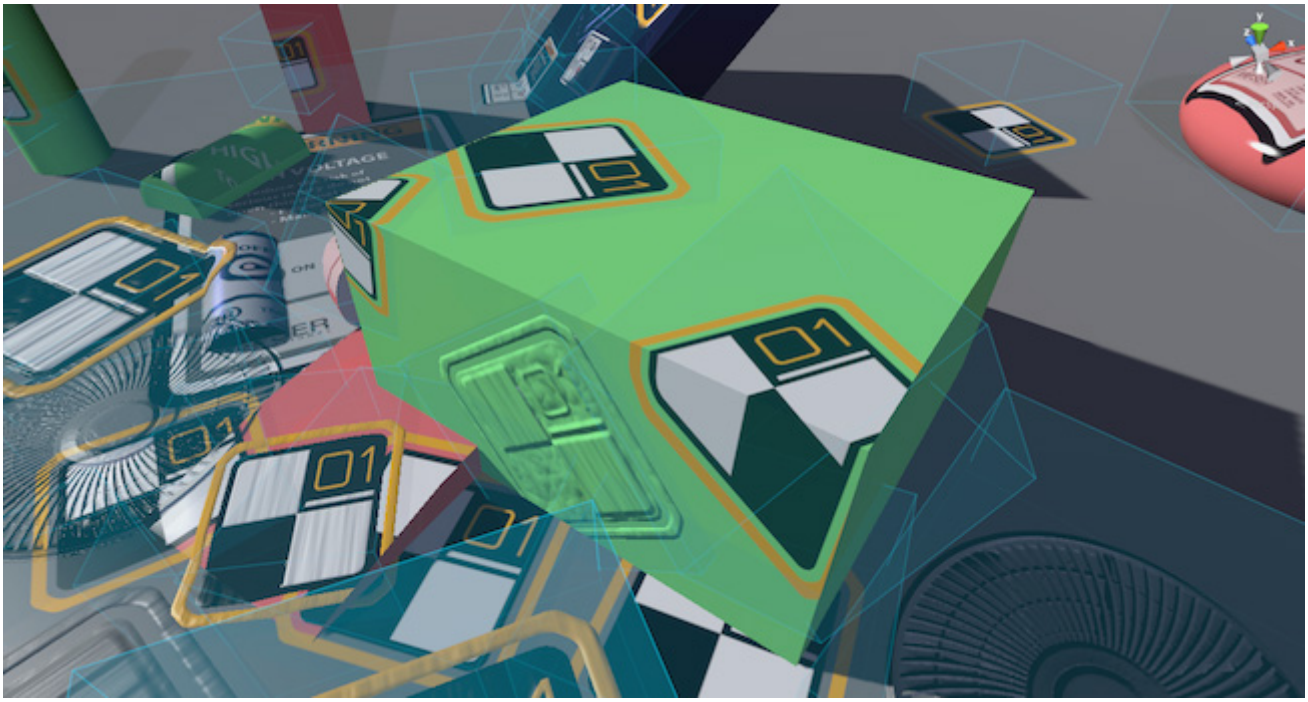
After regular deferred shading light pass is done, a sphere is drawn for each custom light, with a shader that computes illumination and adds it to the lighting buffer.

Decals in Deferred Shading

This scene shows a basic implementation of “deferred decals”.



The idea is: after G-buffer is done, draw each “shape” of the decal (a box) and modify the G-buffer contents. This is very similar to how lights are done in deferred shading, except instead of accumulating the lighting we modify the G-buffer textures.



Each decal is implemented as a box here, and affects any geometry inside the box volume.

GPU instancing

Leave feedback

Introduction

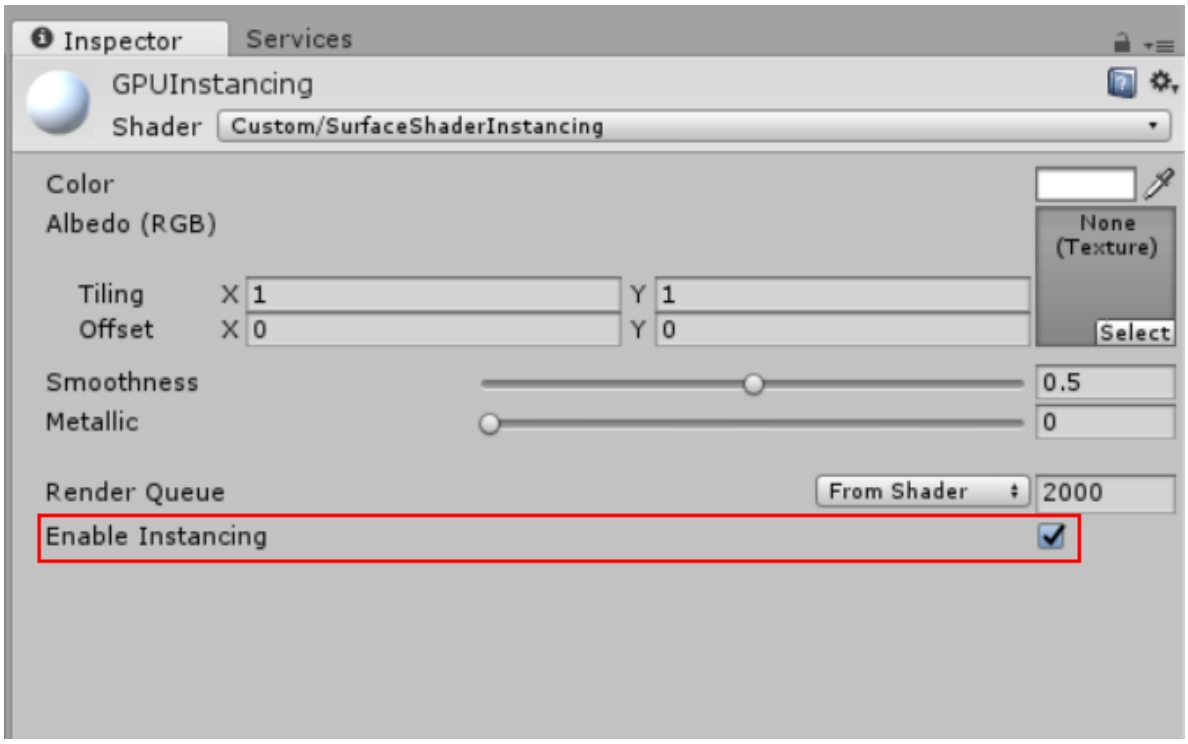
Use GPU Instancing to draw (or render) multiple copies of the same **Mesh** at once, using a small number of draw calls. It is useful for drawing objects such as buildings, trees and grass, or other things that appear repeatedly in a **Scene**.

GPU Instancing only renders identical Meshes with each draw call, but each instance can have different parameters (for example, color or scale) to add variation and reduce the appearance of repetition.

GPU Instancing can reduce the number of draw calls used per Scene. This significantly improves the rendering performance of your project.

Adding instancing to your Materials

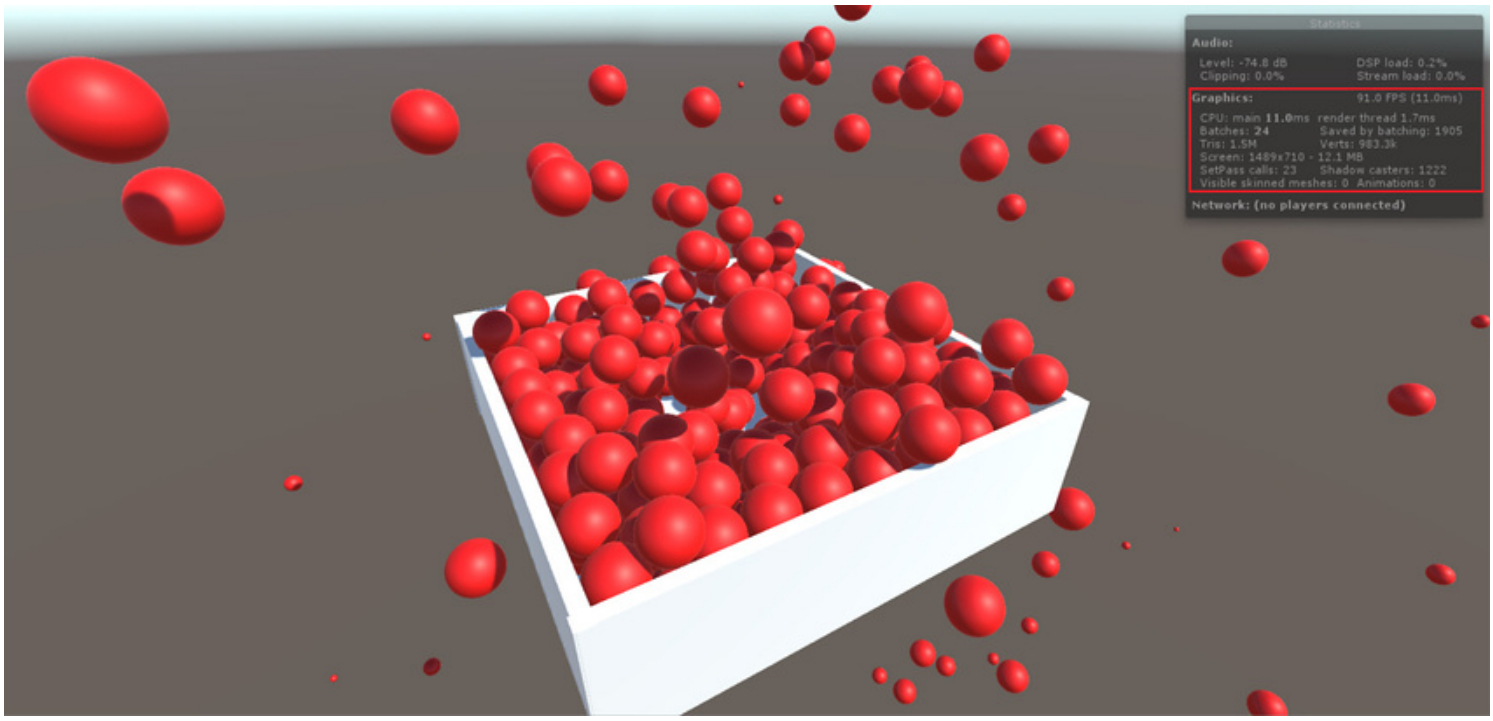
To enable GPU Instancing on Materials, select your Material in the Project window, and in the **Inspector**, tick the **Enable Instancing** checkbox.



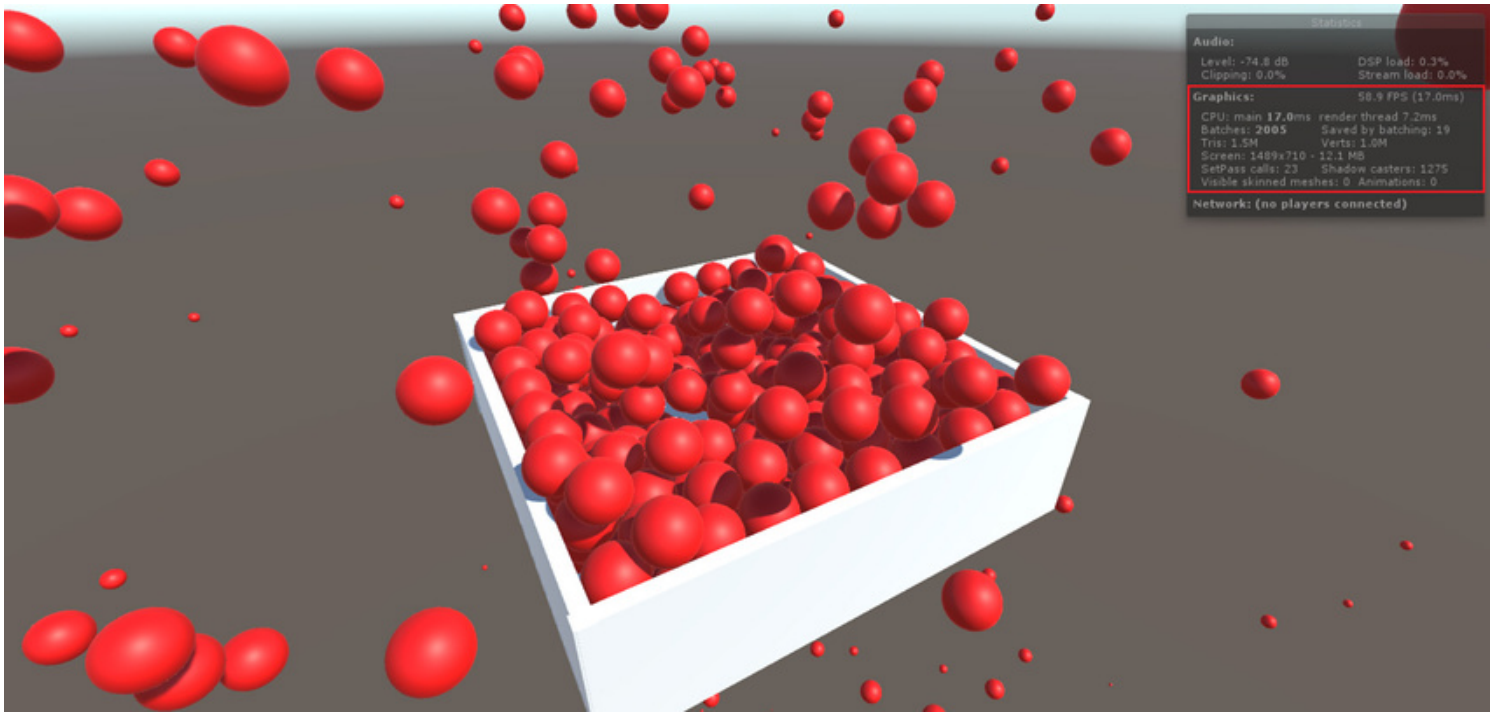
The **Enable Instancing** checkbox as it appears in the Material Inspector window

Unity only displays this checkbox if the Material Shader supports GPU Instancing. This includes Standard, StandardSpecular and all surface Shaders. See documentation on standard Shaders for more information.

The screenshots below show the same Scene with multiple **GameObjects**; in the top image GPU Instancing is enabled, in the bottom image it is not. Note the difference in **FPS**, **Batches** and **Saved by batching**.



With GPU Instancing: A simple Scene that includes multiple identical GameObjects that have GPU Instancing enabled



No GPU Instancing: A simple Scene that includes multiple identical GameObjects that do not have GPU Instancing enabled.

When you use GPU instancing, the following restrictions apply:

Unity automatically picks MeshRenderer **components** and `Graphics.DrawMesh` calls for instancing. Note that SkinnedMeshRenderer is not supported.

Unity only batches GameObjects that share the same Mesh and the same Material in a single GPU instancing draw call. Use a small number of Meshes and Materials for better instancing efficiency. To create variations, modify your shader **scripts** to add per-instance data (see next section to learn more about this).

You can also use the calls Graphics.DrawMeshInstanced and Graphics.DrawMeshInstancedIndirect to perform GPU Instancing from your scripts.

GPU Instancing is available on the following platforms and APIs:

DirectX 11 and **DirectX 12** on Windows

OpenGL Core 4.1+/ES3.0+ on Windows, macOS, Linux, **iOS** and Android

Metal on macOS and iOS

Vulkan on Windows and Android

PlayStation 4 and **Xbox One**

WebGL (requires WebGL 2.0 API)

Adding per-instance data

By default, Unity only batches instances of GameObjects with different Transforms in each instanced draw call. To add more variance to your instanced GameObjects, modify your Shader to add per-instance properties such as Material color.

The example below demonstrates how to create an instanced Shader with different colour values for each instance.

```
Shader "Custom/InstancedColorSurfaceShader" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }

    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light types
        #pragma surface surf Standard fullforwardshadows
        // Use Shader model 3.0 target
        #pragma target 3.0
        sampler2D _MainTex;
        struct Input {
            float2 uv_MainTex;
        };
        half _Glossiness;
        half _Metallic;
        UNITY_INSTANCING_BUFFER_START(Props)
           UNITY_DEFINE_INSTANCED_PROP(fixed4, _Color)
        UNITY_INSTANCING_BUFFER_END(Props)
        void surf (Input IN, inout SurfaceOutputStandard o) {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * UNITY_ACCESS_INSTANCED_PROP(Props,
            o.Albedo = c.rgb;
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
```

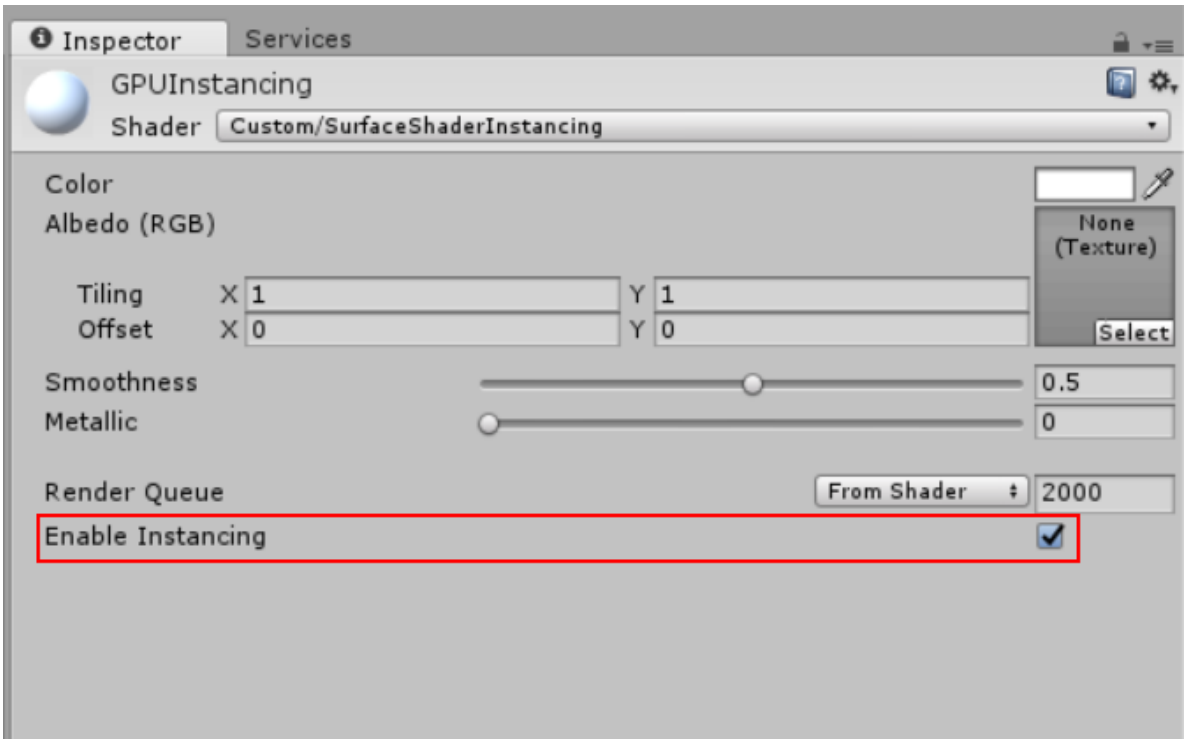
```
        ENDCG
    }
    FallBack "Diffuse"
}
```

When you declare `_Color` as an instanced property, Unity will gather `_Color` values from the MaterialPropertyBlock objects set on GameObjects and put them in a single draw call.

```
MaterialPropertyBlock props = new MaterialPropertyBlock();
MeshRenderer renderer;

foreach (GameObject obj in objects)
{
   float r = Random.Range(0.0f, 1.0f);
   float g = Random.Range(0.0f, 1.0f);
   float b = Random.Range(0.0f, 1.0f);
   props.SetColor("_Color", new Color(r, g, b));

   renderer = obj.GetComponent<MeshRenderer>();
   renderer.SetPropertyBlock(props);
}
```

Note that in normal cases (where an instancing shader is not used, or `_Color` is not a per-instance property), draw call batches are broken due to different values in the MaterialPropertyBlock.

For these changes to take effect, you must enable GPU Instancing. To do this, select your Shader in the Project window, and in the Inspector, tick the **Enable Instancing** checkbox.



The Enable Instancing checkbox as shown in the Shader Inspector window

Adding instancing to vertex and fragment Shaders

The following example takes a simple unlit Shader and makes it capable of instancing with different colors:

```
Shader "SimplestInstancedShader"
{
    Properties
    {
        _Color ("Color", Color) = (1, 1, 1, 1)
    }

    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile_instancing
            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                UNITY_VERTEX_INPUT_INSTANCE_ID
            };
```

```
            struct v2f
            {
                float4 vertex : SV_POSITION;
                UNITY_VERTEX_INPUT_INSTANCE_ID // necessary only if you want to access insta
            };

            UNITY_INSTANCING_BUFFER_START(Props)
                UNITY_DEFINE_INSTANCED_PROP(float4, _Color)
            UNITY_INSTANCING_BUFFER_END(Props)

            v2f vert(appdata v)
            {
                v2f o;

                UNITY_SETUP_INSTANCE_ID(v);
                UNITY_TRANSFER_INSTANCE_ID(v, o); // necessary only if you want to access in

                o.vertex = UnityObjectToClipPos(v.vertex);
                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                UNITY_SETUP_INSTANCE_ID(i); // necessary only if any instanced properties ar
                return UNITY_ACCESS_INSTANCED_PROP(Props, _Color);
            }
            ENDCG
        }
    }
}
```

Shader modifications

Addition	Function
`#pragma multi_compile_instancing`	Use this to instruct Unity to generate instancing variants. It is not necessary for **surface Shaders**.
`UNITY_VERTEX_INPUT_INSTANCE_ID`	Use this in the **vertex Shader** input/output structure to define an instance ID. See `SV_InstanceID` for more information.
`UNITY_INSTANCING_BUFFER_START(name)` / `UNITY_INSTANCING_BUFFER_END(name)`	Every per-instance property must be defined in a specially named constant buffer. Use this pair of macros to wrap the properties you want to be made unique to each instance.
`UNITY_DEFINE_INSTANCED_PROP(float4, _Color)`	Use this to define a per-instance Shader property with a type and a name. In this example, the `_Color` property is unique.
`UNITY_SETUP_INSTANCE_ID(v);`	Use this to make the instance ID accessible to Shader functions. It must be used at the very beginning of a vertex Shader, and is optional for **fragment Shaders**.
`UNITY_TRANSFER_INSTANCE_ID(v, o);`	Use this to copy the instance ID from the input structure to the output structure in the vertex Shader. This is only necessary if you need to access per-instance data in the fragment Shader.

Addition	Function
`UNITY_ACCESS_INSTANCED_PROP(arrayName, color)`	Use this to access a per-instance Shader property declared in an instancing constant buffer. It uses an instance ID to index into the instance data array. The `arrayName` in the macro must match the one in `UNITY_INSTANCING_BUFFER_END(name)` macro.

Notes:

When using multiple per-instance properties, you don't need to fill all of them in `MaterialPropertyBlocks`.

If one instance lacks the property, Unity takes the default value from the referenced Material. If the material does not have a default value for the specified property, Unity sets the value to 0. Do not put non-instanced properties in the `MaterialPropertyBlock`, because this disables instancing. Instead, create different Materials for them.

Advanced GPU instancing tips

Batching priority

When batching, Unity prioritizes Static batching over instancing. If you mark one of your GameObjects for static batching, and Unity successfully batches it, Unity disables instancing on that GameObject, even if its Renderer uses an instancing Shader. When this happens, the Inspector window displays a warning message suggesting that you disable Static Batching. To do this, open the Player Settings (**Edit** > **Project Settings** > **Player**), open **Other Settings**, and under the **Rendering** section, untick the **Static Batching** checkbox.

Unity prioritizes instancing over **dynamic batching**. If Unity can instance a Mesh, it disables **dynamic batching** for that Mesh.

Graphics.DrawMeshInstanced

Some factors can prevent GameObjects from being instanced together automatically. These factors include Material changes and depth sorting. Use Graphics.DrawMeshInstanced to force Unity to draw these objects using GPU instancing. Like Graphics.DrawMesh, this function draws Meshes for one frame without creating unnecessary GameObjects.

Graphics.DrawMeshInstancedIndirect

Use `DrawMeshInstancedIndirect` in a script to read the parameters of instancing draw calls, including the number of instances, from a compute buffer. This is useful if you want to populate all of the instance data from the GPU, and the CPU does not know the number of instances to draw (for example, when performing GPU culling). See API documentation on Graphics.DrawMeshInstancedIndirect for a detailed explanation and code examples.

Global Illumination support

Since Unity 2018.1, Global Illumination (GI) rendering is supported by GPU Instancing in the form of light probes, occlusion probes (in Shadowmask mode) and lightmap STs. Standard shaders and surface shaders have GI support automatically enabled.

Dynamic renderers affected by light probes and occlusion probes baked in the scene, and static renderers baked to the same lightmap texture, can be automatically batched together using GPU Instancing by Forward and Deferred render loop.

For Graphics.DrawMeshInstanced, you can enable light probe and occlusion probe rendering by setting the LightProbeUsage argument to CustomProvided and providing a MaterialPropertyBlock with probe data copied in. See API documentation on LightProbes.CalculateInterpolatedLightAndOcclusionProbes for a detailed explanation and code examples.

Global Illumination and GPU Instancing

GPU Instancing supports Global Illumination (GI) rendering in Unity. Each GPU instance can support GI coming from either different Light Probes, one lightmap (but multiple atlas regions in that lightmap), or one Light Probe Proxy Volume component (baked for the space volume containing all the instances). Standard shaders and surface shaders come with this support enabled.

You can use GPU Instancing to automatically batch dynamic Mesh Renderers affected by baked Light Probes (including their occlusion data), or static Mesh Renderers baked to the same lightmap Texture, via a Forward and Deferred render loop. See documentation on the Rendering pipeline for more information.

For Graphics.DrawMeshInstanced, you can enable the rendering of Light Probes (including their occlusion data) by setting the LightProbeUsage argument to CustomProvided and providing a MaterialPropertyBlock with probe data copied in. See API documentation on LightProbes.CalculateInterpolatedLightAndOcclusionProbes for a detailed explanation and code examples.

Alternatively, you can pass an LPPV component reference and LightProbeUsage.UseProxyVolume to `Graphics.DrawMeshInstanced`. When you do this, all instances sample the volume for the L0 and L1 bands of the Light Probe data. Use `MaterialPropertyBlock` if you want to supplement L2 data and occlusion data. For more information, see Light Probes: Technical Information.

Shader warming-up

Since Unity 2017.3, you need to warm up shaders to use instancing on OpenGL if you want absolutely smooth rendering when the shader renders for the first time. If you warm up shaders for instancing on a platform that doesn't require shader warm up, nothing will happen.

See ShaderVariantCollection.WarmUp and Shader.WarmupAllShaders for more information.

#pragma instancing_options

The `#pragma instancing_options` directive can use the following switches:

Switch	Function
`forcemaxcount:batchSize` and `maxcount:batchSize`	On most platforms, Unity automatically calculates the instancing data array size by dividing the maximum constant buffer size on the target device with the size of the structure containing all per-instance properties. Generally you don't need to worry about the batch size. However, on some platforms (Vulkan, Xbox One and Switch), a fixed array size is still required. You can specify the batch size for those platforms by using `maxcount` option. The option is completely ignored on the other platforms. If you really want to force a batch size for all platforms, use `forcemaxcount` (for example, when you know you will only issue draws with 256 instanced sprites via DrawMeshInstanced). The default value for the two options is 500.
`assumeuniformscaling`	Use this to instruct Unity to assume that all the instances have uniform scalings (the same scale for all X, Y and Z axes).
`nolodfade`	Use this to prevent Unity from applying GPU Instancing to LOD fade values.
`nolightprobe`	Use this to prevent Unity from applying GPU Instancing to Light Probe values (including their occlusion data). This is useful for performance if you are absolutely sure that there are no GameObjects using both GPU Instancing and Light Probes.
`nolightmap`	Use this to prevent Unity from applying GPU Instancing to Lightmap ST (atlas information) values. This is useful for performance if you are absolutely sure that there are no GameObjects using both GPU Instancing and lightmaps.
`procedural:FunctionName`	Use this to instruct Unity to generate an additional variant for use with Graphics.DrawMeshInstancedIndirect. At the beginning of the vertex Shader stage, Unity calls the function specified after the colon. To set up the instance data manually, add per-instance data to this function in the same way you would normally add per-instance data to a Shader. Unity also calls this function at the beginning of a fragment Shader if any of the fetched instance properties are included in the fragment Shader.

UnityObjectToClipPos

When writing shader scripts, always use `UnityObjectToClipPos(v.vertex)` instead of `mul(UNITY_MATRIX_MVP,v.vertex)`.

While you can continue to use `UNITY_MATRIX_MVP` as normal in instanced Shaders, `UnityObjectToClipPos` is the most efficient way to transform vertex positions from object space into clip space. Unity also implements a Shader upgrader that scans all your Shaders in the project, and automatically replaces any occurrence of `mul(UNITY_MATRIX_MVP, v)` with `UnityObjectToClipPos(v)`.

The console window (menu: **Window** > **General** > **Console**) displays performance warnings if there are still places where `UNITY_MATRIX_MVP` (along with `UNITY_MATRIX_MV`) is used.

Further notes

Surface Shaders have instancing variants generated by default, unless you specify `noinstancing` in the `#pragma` surface directive. Standard and StandardSpecular Shaders are already modified to have instancing support, but with no per-instance properties defined other than the transforms. Unity ignores uses of `#pragma multi_compile_instancing` in a surface Shader.

Unity strips instancing variants if GPU Instancing is not enabled on any GameObject in the Scene. To override the stripping behaviour, open the Graphics Settings (menu: **Edit** > **Project Settings** > **Graphics**), navigate to the **Shader stripping** section and change the **Instancing Variants**.

For `Graphics.DrawMeshInstanced`, you need to enable GPU Instancing on the Material that the script is passing into this method. However, `Graphics.DrawMeshInstancedIndirect` does not require you to enable GPU Instancing. The indirect instancing keyword `PROCEDURAL_INSTANCING_ON` is not affected by stripping.

Instanced draw calls appear in the Frame Debugger as **Draw Mesh (instanced)**.

You don't always need to define per-instance properties. However, setting up an instance ID is mandatory, because world matrices need it to function correctly. Surface shaders automatically set up an instance ID. You must set up the instance ID for Custom Vertex and Fragment shaders manually. To do this, use `UNITY_SETUP_INSTANCE_ID` at the beginning of the Shader.

When using forward rendering, Unity cannot efficiently instance objects that are affected by multiple lights. Only the base pass can make effective use of instancing, not the added passes. For more information about lighting passes, see documentation on Forward Rendering and Pass Tags

If you have more than two passes for multi-pass Shaders, only the first passes can be instanced. This is because Unity forces the later passes to be rendered together for each object, forcing Material changes.

All the Shader macros used in the above examples are defined in *UnityInstancing.cginc*. Find this file in the following directory: *[Unity installation folder]\Editor\Data\CGIncludes*.

2017–10–24 Page amended with editorial review

Enable instancing checkbox guidance, DrawMeshInstancedIndirect, #pragma multi-compile added in 5.6

Shader warm up for GPU instancing added in 2017.3

Global Illumination (GI) support in GPU instancing added in 2018.1

Sparse Textures

Leave feedback

Sparse textures (also known as “tiled textures” or “mega-textures”) are textures that are too large to fit in graphic memory in their entirety. To handle them, Unity breaks the main texture down into smaller rectangular sections known as “tiles”. Individual tiles can then be loaded as necessary. For example, if the **camera** can only see a small area of a sparse texture, then only the tiles that are currently visible need to be in memory.

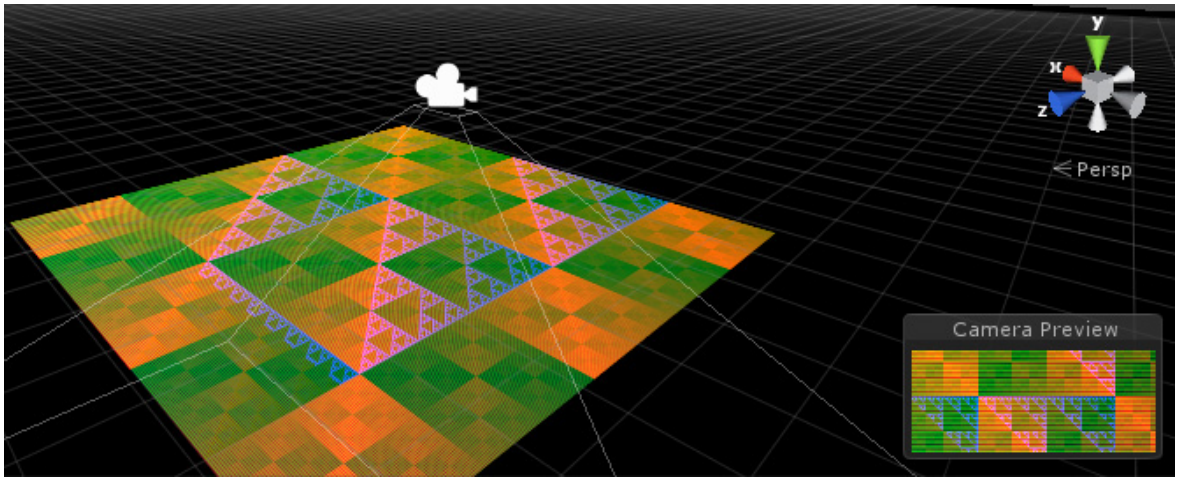
Aside from the tiling, a sparse texture behaves like any other texture in usage. **Shaders** can use them without special modification and they can have mipmaps, use all texture filtering modes, **etc**. If a particular tile cannot be loaded for some reason then the result is undefined; some GPUs show a black area where the missing tile should be but this behaviour is not standardised.

Not all hardware and platforms support sparse textures. For example, on DirectX systems they require DX11.2 (Windows 8.1) and a fairly recent GPU. On OpenGL they require ARB_sparse_texture extension support. Sparse textures only support non-compressed **texture formats**.

See the SparseTexture script reference page for further details about handling sparse textures from **scripts**.

Example Project

A minimal example project for sparse textures is available here.



Sparse texture as shown in the example project

The example shows a simple procedural texture pattern and lets you move the camera to view different parts of it. Note that the project requires a recent GPU and a DirectX 11.2 (Windows 8.1) system, or using OpenGL with *ARB_sparse_texture* support.

Graphics hardware capabilities and emulation

Leave feedback

The graphics hardware that ultimately renders a **Scene** is controlled by specialised programs called Shaders. The capabilities of the hardware have improved over time, and the general set of features that were introduced with each phase is known as a Shader Model. Successive Shader Models have added support for longer programs, more powerful branching instructions and other features, and these have enabled improvements in the graphics of games.

The Unity Editor supports emulation of several sets of Shader Models and graphics API restrictions, for getting a quick overview of how the game might look like when running on a particular GPU or graphics API. Note that the in-editor emulation is very approximate, and it is always advisable to actually run the game build on the hardware you are targeting.

To choose the graphics emulation level, go to the **Edit** > **Graphics Emulation** menu. Note that the available options change depending on the platform you are currently targeting in the Build Settings. You can restore the full capabilities of your hardware by choosing **No Emulation**. If your development computer doesn't support a particular Shader Model then the menu entry will be disabled.

Shader Model 4 (Standalone & Universal Windows Platform)

Emulates DirectX 10 feature set (PC GPUs made during 2007–2009).
Turns off support for compute Shaders and related features (compute buffers, random-write Textures), sparse Textures, and tessellation Shaders.

Shader Model 3 (Standalone platform)

Emulates DirectX 9 SM3.0 feature set (PC GPUs made during 2004–2006).
In addition to features turned off by Shader Model 4 emulation, this also turns off support for draw call instancing, Texture Arrays, and geometry Shaders. It enforces a maximum of 4 simultaneous render targets, and a maximum of 16 Textures used in a single Shader.

Shader Model 2 (Standalone platform)

Emulates DirectX 9 SM2.0 feature set (PC GPUs made during 2002–2004).
In addition to features turned off by Shader Model 3 emulation, this also turns off support for HDR rendering, Linear color space and depth Textures.

OpenGL ES 3.0 (Android platform)

Emulates mobile OpenGL ES 3.0 feature set.
Turns off support for compute Shaders and related features (compute buffers, random-write Textures), sparse Textures, tessellation Shaders and geometry Shaders. Enforces maximum of 4 simultaneous render targets, and maximum of 16 Textures used in a single Shader. Maximum allowed Texture size is set to 4096, and maximum **cubemap** size to 2048. Realtime soft shadows are disabled.

Metal (iOS, tvOS platforms)

Emulates mobile Metal feature set.
Same restrictions applied as GLES3.0 emulation, except that the maximum cubemap size is set to 4096.

OpenGL ES 2.0 (Android, iOS, tvOS platforms)

Emulates mobile OpenGL ES 2.0 feature set.
In addition to features turned off by GLES3.0 emulation, this also turns off support for draw call instancing, Texture arrays, 3D Textures and multiple render targets. Enforces a maximum of 8 Textures used in a single Shader. Maximum allowed cubemap size is set to 1024.

WebGL 1 and WebGL 2 (WebGL platform)

Emulates typical WebGL graphics restrictions.
Very similar to GLES2.0 and GLES3.0 emulation levels above, except that supported Texture sizes are higher (8192 for regular Textures, 4096 for cubemaps), and 16 Textures are allowed in a single Shader.

Shader Model 2 - DX11 FL9.3 (Universal Windows Platform)

Emulates typical Windows Phone graphics feature set.
Very similar to Shader Model 2 emulation, but also disables multiple render targets and separate alpha blending.

- 2017–05–16 Page amended with no editorial review

CullingGroup API

Leave feedback

CullingGroup offers a way to integrate your own systems into Unity's culling and **LOD** pipeline. This can be used for many purposes; for example:

Simulating a crowd of people, while only having full **GameObjects** for the characters that are actually visible right now
Building a GPU particle system driven by Graphics.DrawProcedural, but skipping **rendering particle systems** that are behind a wall
Tracking which spawn points are hidden from the **camera** in order to spawn enemies without the player seeing them 'pop' into view
Switching characters from full-quality animation and AI calculations when close, to lower-quality cheaper behaviour at a distance
Having 10,000 marker points in your **scene** and efficiently finding out when the player gets within 1m of any of them

The API works by having you provide an array of bounding spheres. These visibility of these spheres relative to a particular camera is then calculated, along with a 'distance band' value that can be treated like a LOD level number.

Getting Started with CullingGroup

There are no **components** or visual tools for working with CullingGroups; they are purely accessible via script.

A CullingGroup can be constructed using the 'new' operator:

```
CullingGroup group = new CullingGroup();
```

To have the CullingGroup perform visibility calculations, specify the camera it should use:

```
group.targetCamera = Camera.main;
```

Create and populate an array of BoundingSphere structures with the positions and radii of your spheres, and pass it to SetBoundingSpheres along with the number of spheres that are actually in the array. The number of spheres does not need to be the same as the length of the array; in fact we recommend creating an array that is big enough to hold the most spheres you will ever have at one time, even if initially the number of spheres you actually have in the array is very low. This way you avoid having to resize the array as spheres are added or removed, which is an expensive operation.

```
BoundingSphere[] spheres = new BoundingSphere[1000];
spheres[0] = new BoundingSphere(Vector3.zero, 1f);
group.SetBoundingSpheres(spheres);
group.SetBoundingSphereCount(1);
```

At this point, the CullingGroup will begin computing the visibility of the single sphere each frame.

To clean up the CullingGroup and free all memory it uses, dispose of the CullingGroup via the standard .NET IDisposable mechanism:

```
group.Dispose();
group = null;
```

Recieving results via the onStateChanged callback

The most efficient way to respond to spheres changing their visibility or distance state is to use the onStateChanged callback field. Set this to a function which takes a CullingGroupEvent structure as an argument; it will then be called after culling is complete, for each sphere that has changed state. The members of the CullingGroupEvent structure tell you about the previous and new states of the sphere.

```
group.onStateChanged = StateChangedMethod;

private void StateChangedMethod(CullingGroupEvent evt)
{
    if(evt.hasBecomeVisible)
        Debug.LogFormat("Sphere {0} has become visible!", evt.index);
    if(evt.hasBecomeInvisible)
        Debug.LogFormat("Sphere {0} has become invisible!", evt.index);
}
```

Recieving results via the CullingGroup Query API

In addition to the onStateChanged delegate, the CullingGroup provides an API for retrieving the latest visibility and distance results of any sphere in the bounding spheres array. To check the states of a single sphere, use the IsVisible and GetDistance methods:

```
bool sphereIsVisible = group.IsVisible(0);
int sphereDistanceBand = group.GetDistance(0);
```

To check the states of multiple spheres, you can use the QueryIndices method. This method scans a continuous range of spheres to find ones that match a given visibility or distance state.

```
// Allocate an array to hold the resulting sphere indices - the size of the array determ
int[] resultIndices = new int[1000];
// Also set up an int for storing the actual number of results that have been placed int
int numResults = 0;

// Find all spheres that are visible
```

```
numResults = group.QueryIndices(true, resultIndices, 0);
// Find all spheres that are in distance band 1
numResults = group.QueryIndices(1, resultIndices, 0);
// Find all spheres that are hidden in distance band 2, skipping the first 100
numResults = group.QueryIndices(false, 2, resultIndices, 100);
```

Remember that the information retrieved by the query API is only updated when the camera used by the CullingGroup actually performs its culling.

CullingGroup API Best Practices

When considering how you might apply CullingGroup to your project, consider the following aspects of the CullingGroup design.

Using visibility

All the volumes for which CullingGroup computes visibility are defined by bounding spheres - in practice, a position (the center of the sphere) and a radius value. No other bounding shapes are supported, for performance reasons. In practice this means you will be defining a sphere that fully encloses the object you are interested in culling. If a tighter fit is needed, consider using multiple spheres to cover different parts of the object, and making decisions based on the visibility state of all of the spheres.

In order to evaluate visibility, the CullingGroup needs to know which camera visibility should be computed from. Currently a single CullingGroup only supports a single camera. If you need to evaluate visibility to multiple cameras, you should use one CullingGroup per camera and combine the results.

The CullingGroup will calculate visibility based on frustum culling and static **occlusion culling** only. It will not take dynamic objects into account as potential occluders.

Using distance

The CullingGroup is capable of calculating the distance between some reference point (for example, the position of the camera or player) and the closest point on each sphere. This distance value is not provided to you directly, but is instead quantized using a set of threshold values that you provide, in order to calculate a discrete 'distance band' integer result. The intention is that you interpret these distance bands as 'close range', 'medium range', 'far range', and so on.

The CullingGroup will provide callbacks when an object moves from being in one band to being in another, giving you the opportunity to do things like change the behaviour of that object to something less CPU-intensive.

Any spheres that are beyond the last distance band will be considered to be invisible, allowing you to easily construct a culling implementation which completely deactivates objects that are very far away. If you do not want this behaviour, simply set your final threshold value to be at an infinite distance away.

Only a single reference point is supported per CullingGroup.

Performance and design

The CullingGroup API does not give you the ability to make changes to your scene and then immediately request the new visibility state of a bounding sphere. For performance reasons, the CullingGroup only calculates new visibility information during execution of culling for the camera as a whole; it's at this point that the information is available to you, either via a callback, or via the CullingGroup query API. In practice, this means you should approach CullingGroup in an asynchronous manner.

The bounding spheres array you provide to the CullingGroup is referenced by the CullingGroup, rather than copied. This means you should keep a reference to the array that you pass to SetBoundingSpheres, and that you can modify the contents of this array without needing to call SetBoundingSpheres again. If you need multiple CullingGroups that calculate visibility and distances for the same set of spheres - for example, for multiple cameras - then it's efficient to have all the CullingGroups share the same bounding sphere array instance.

Asynchronous Texture Upload

Leave feedback

Asynchronous Texture Upload enables asynchronous loading of Texture Data from disk and enables time-sliced upload to GPU on the Render-thread. This reduces wait for GPU uploads in the main thread. Async Texture Upload will automatically be used for all Textures that are not read-write enabled, so to use this feature no direct action is required. You can however control some aspects of how the async upload operates, and so some understanding of the process is useful to be able to use these controls.

When the project is built, the texture data of asynchronous uploadable textures are stored in as streaming resource files and are loaded asynchronously.

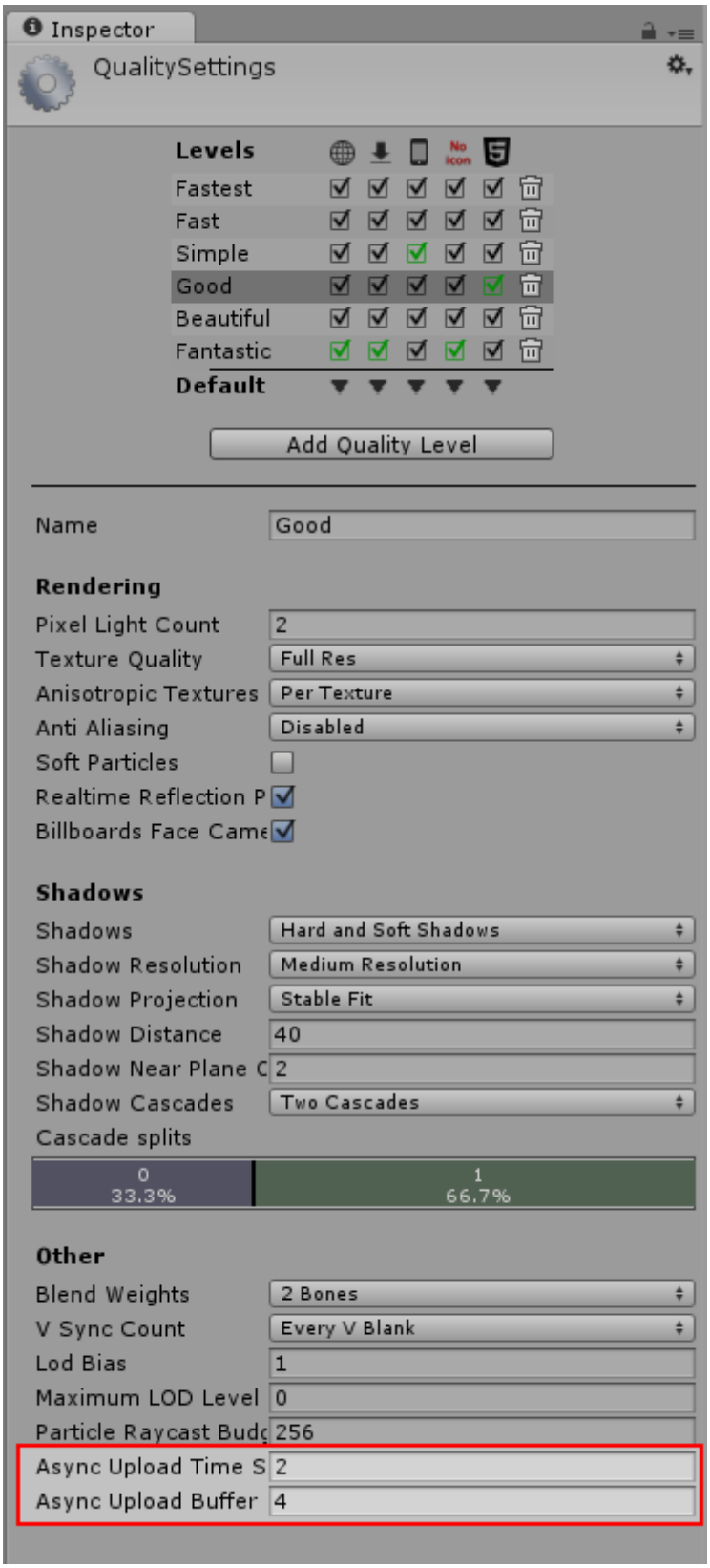
Simple & Full Control Over Memory / Time-Slicing

A single ring-buffer is reused to load the texture data and upload it to the GPU, which reduces the amount of memory allocations required. For example, if you have 20 small textures, Unity will set up an asynchronous load request for those 20 textures in one go. If you have one huge texture, Unity will request only one.

If the buffer size is not large enough for the textures being requested, it will automatically resize to accomodate, however it is always optimal to try to set the size to fit the largest sized texture that you will be uploading from the outset, so that the buffer does not need to resize for each new larger texture it encounters.

The time spent on texture upload each frame can be controlled, with larger values meaning the textures will become ready on the GPU sooner but with the overhead of more CPU time being used during those frames for other processing. This CPU time is only used if there are textures waiting in the buffer to be uploaded to the GPU.

The size of the buffer and time-slice can be specified through the Quality Settings panel:



The Async Upload settings in the Quality Settings panel

Async Texture Upload Scripting API

We provide the ability to control the Buffer Size and the Time-Slice value from script.

Time-Slice

See Script Ref: QualitySettings.asyncUploadTimeSlice.

Sets the Time-Slice in milliseconds for CPU time spent on Asynchronous Texture Uploads per frame. Depending on the target platform and API, you may want to set this. Time is only spent on the function call if there are textures to upload, otherwise it early-exits.

Buffer Size

See Script Ref: QualitySettings.asyncUploadBufferSize

Set the Ring Buffer Size for Asynchronous Texture Uploads. The size is in mega-bytes. Ensure that you set a reasonable size depending on the Target platform. Also please ensure that it is always sufficient to load any huge texture in your games. For example if you have a **Cubemap** of size 22MB and if you set the size of the RingBuffer to 16MB, the App will automatically resize the Ringbuffer to 22MB while loading that **scene**.

Notes

For non-read/write enabled textures, the TextureData is part of resS (Streaming Resource) and upload now happens on Render-Thread. Availability of Texture is guaranteed during call to AwakeFromLoad just as before, so there are no changes in terms of order of loading or availability of Textures on **Rendering**.

For other types of texture loading, such as read/write enabled textures, textures loaded directly with the LoadImage(byte[] data) function, or loading from the Resources folder, the Asynchronous buffer loading is not used - the older Synchronous method is used.

Procedural Mesh Geometry

Leave feedback

The Mesh class gives script access to an object's mesh geometry, allowing meshes to be created or modified at runtime. This technique is useful for graphical effects (eg, stretching or squashing an object) but can also be useful in level design and optimisation. The following sections explain the basic details of how a mesh is constructed along with an exploration of the API and an example.

Anatomy of a Mesh.
Using the Mesh Class.

See Also

Mesh scripting class.
Importing Meshes.

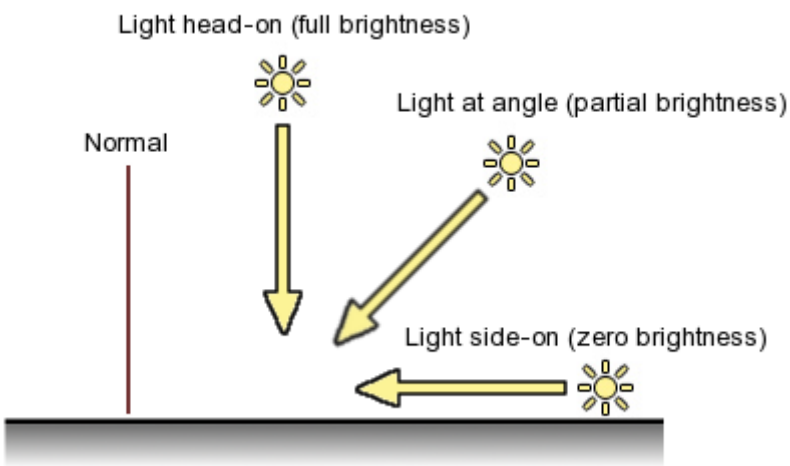
Anatomy of a Mesh

Leave feedback

A mesh consists of triangles arranged in 3D space to create the impression of a solid object. A triangle is defined by its three corner points or vertices. In the Mesh class, the vertices are all stored in a single array and each triangle is specified using three integers that correspond to indices of the vertex array. The triangles are also collected together into a single array of integers; the integers are taken in groups of three from the start of this array, so elements 0, 1 and 2 define the first triangle, 3, 4 and 5 define the second, and so on. Any given vertex can be reused in as many triangles as desired but there are reasons why you may not want to do this, as explained below.

Lighting and Normals

The triangles are enough to define the basic shape of the object but extra information is needed to display the mesh in most cases. To allow the object to be shaded correctly for lighting, a normal vector must be supplied for each vertex. A normal is a vector that points outward, perpendicular to the mesh surface at the position of the vertex it is associated with. During the shading calculation, each vertex normal is compared with the direction of the incoming light, which is also a vector. If the two vectors are perfectly aligned, then the surface is receiving light head-on at that point and the full brightness of the light will be used for shading. A light coming exactly side-on to the normal vector will give no illumination to the surface at that position. Typically, the light will arrive at an angle to the normal and so the shading will be somewhere in between full brightness and complete darkness, depending on the angle.



Since the mesh is made up of triangles, it may seem that the normals at corners will simply be perpendicular to the plane of their triangle. However, normals are actually interpolated across the triangle to give the surface direction of the intermediate positions between the corners. If all three normals are pointing in the same direction then the triangle will be uniformly lit all over. The effect of having separate triangles uniformly shaded is that the edges will be very crisp and distinct. This is exactly what is required for a model of a cube or other sharp-edged solid but the **interpolation** of the normals can be used to create smooth shading to approximate a curved surface.

To get crisp edges, it is necessary to double up vertices at each edge since both of the two adjacent triangles will need their own separate normals. For curved surfaces, vertices will usually be shared along edges but a bit of intuition is often required to determine the best direction for the shared normals. A normal might simply be the average of the normals of the planes of the surrounding triangles. However, for an object like a sphere, the normals should just be pointing directly outward from the sphere's centre.

By calling Mesh.RecalculateNormals, you can get Unity to work out the normals' directions for you by making some assumptions about the "meaning" of the mesh geometry; it assumes that vertices shared between triangles indicate a smooth surface while doubled-up vertices indicate a crisp edge. While this is not a bad approximation in most cases, RecalculateNormals will be tripped up by some texturing situations where vertices must be doubled even though the surface is smooth.

Texturing

In addition to the lighting, a model will also typically make use of texturing to create fine detail on its surface. A texture is a bit like an image printed on a stretchable sheet of rubber. For each mesh triangle, a triangular area of the texture image is defined and that texture triangle is stretched and "pinned" to fit the mesh triangle. To make this work, each vertex needs to store the coordinates of the image position that will be pinned to it. These coordinates are two dimensional and scaled to the 0..1 range (0 means the bottom/left of the image and 1 means the right/top). To avoid confusing these coordinates with the Cartesian coordinates of the 3D world, they are referred to as U and V rather than the more familiar X and Y, and so they are commonly called UV coordinates.

Like normals, texture coordinates are unique to each vertex and so there are situations where you need to double up vertices purely to get different UV values across an edge. An obvious example is where two adjacent triangles use discontinuous parts of the texture image (eyes on a face texture, say). Also, most objects that are fully enclosed volumes will need a "seam" where an area of texture wraps around and joins together. The UV values at one side of the seam will be different from those at the other side.

See Also

Using the Mesh Class page.
Mesh scripting class reference.

Using the Mesh Class

Leave feedback

The Mesh class is the basic script interface to an object's mesh geometry. It uses arrays to represent the vertices, triangles, normals and texture coordinates and also supplies a number of other useful properties and functions to assist mesh generation.

Accessing an Object's Mesh

The mesh data is attached to an object using the **Mesh Filter component** (and the object will also need a Mesh Renderer to make the geometry visible). This component is accessed using the familiar GetComponent function:-

```
var mf: MeshFilter = GetComponent.<MeshFilter>();
// Use mf.mesh to refer to the mesh itself.
```

Adding the Mesh Data

The Mesh object has properties for the vertices and their associated data (normals and UV coordinates) and also for the triangle data. The vertices may be supplied in any order but the arrays of normals and UVs must be ordered so that the indices all correspond with the vertices (ie, element 0 of the normals array supplies the normal for vertex 0, etc). The vertices are Vector3s representing points in the object's local space. The normals are normalised Vector3s representing the directions, again in local coordinates. The UVs are specified as Vector2s, but since the Vector2 type doesn't have fields called U and V, you must mentally convert them to X and Y respectively.

The triangles are specified as triples of integers that act as indices into the vertex array. Rather than use a special class to represent a triangle the array is just a simple list of integer indices. These are taken in groups of three for each triangle, so the first three elements define the first triangle, the next three define the second triangle, and so on. An important detail of the triangles is the ordering of the corner vertices. They should be arranged so that the corners go around clockwise as you look down on the visible outer surface of the triangle, although it doesn't matter which corner you start with.

See Also

Anatomy of a Mesh page.
Mesh scripting class reference.

Example - Creating a Quad

Leave feedback

Unity comes with the **Plane** and **Quad** primitive objects to represent flat surfaces (see the Primitive Objects page for further details). However, it is useful to examine how a minimal quadrilateral **mesh** can be constructed, since this is probably the simplest useful example, with just four vertices for the corners and two triangles.

The first thing is to set the vertices array. We'll assume that the plane lies in the X and Y axes and let its width and height be determined by parameter variables. We'll supply the vertices in the order bottom-left, bottom-right, top-left, top-right.

```
var vertices: Vector3[] = new Vector3[4];

vertices[0] = new Vector3(0, 0, 0);
vertices[1] = new Vector3(width, 0, 0);
vertices[2] = new Vector3(0, height, 0);
vertices[3] = new Vector3(width, height, 0);

mesh.vertices = vertices;
```

(Since the Mesh data properties execute code behind the scenes, it is much more efficient to set up the data in your own array and then assign this to a property rather than access the property array element by element.)

Next come the triangles. Since we want two triangles, each defined by three integers, the triangles array will have six elements in total. Remembering the clockwise rule for ordering the corners, the lower left triangle will use 0, 2, 1 as its corner indices, while the upper right one will use 2, 3, 1.

```
var tri: int[] = new int[6];

//  Lower left triangle.
tri[0] = 0;
tri[1] = 2;
tri[2] = 1;

//  Upper right triangle.
tri[3] = 2;
tri[4] = 3;
tri[5] = 1;

mesh.triangles = tri;
```

A mesh with just the vertices and triangles set up will be visible in the editor but will not look very convincing since it is not correctly shaded without the normals. The normals for the flat plane are very simple - they are all identical and point in the negative Z direction in the plane's local space. With the normals added, the plane will be correctly shaded but remember that you need a light in the **scene** to see the effect.

```
var normals: Vector3[] = new Vector3[4];

normals[0] = -Vector3.forward;
normals[1] = -Vector3.forward;
normals[2] = -Vector3.forward;
normals[3] = -Vector3.forward;

mesh.normals = normals;
```

Finally, adding texture coordinates to the mesh will enable it to display a material correctly. Assuming we want to show the whole image across the plane, the UV values will all be 0 or 1, corresponding to the corners of the texture.

```
var uv: Vector2[] = new Vector2[4];

uv[0] = new Vector2(0, 0);
uv[1] = new Vector2(1, 0);
uv[2] = new Vector2(0, 1);
uv[3] = new Vector2(1, 1);

mesh.uv = uv;
```

The complete script might look a bit like this:-

```
var width: float;
var height: float;

function Start() {
    var mf: MeshFilter = GetComponent.<MeshFilter>();
    var mesh = new Mesh();
    mf.mesh = mesh;

    var vertices: Vector3[] = new Vector3[4];

    vertices[0] = new Vector3(0, 0, 0);
    vertices[1] = new Vector3(width, 0, 0);
    vertices[2] = new Vector3(0, height, 0);
    vertices[3] = new Vector3(width, height, 0);

    mesh.vertices = vertices;

    var tri: int[] = new int[6];

    tri[0] = 0;
    tri[1] = 2;
    tri[2] = 1;

    tri[3] = 2;
    tri[4] = 3;
    tri[5] = 1;

    mesh.triangles = tri;

    var normals: Vector3[] = new Vector3[4];
```

```
    normals[0] = -Vector3.forward;
    normals[1] = -Vector3.forward;
    normals[2] = -Vector3.forward;
    normals[3] = -Vector3.forward;

    mesh.normals = normals;

    var uv: Vector2[] = new Vector2[4];

    uv[0] = new Vector2(0, 0);
    uv[1] = new Vector2(1, 0);
    uv[2] = new Vector2(0, 1);
    uv[3] = new Vector2(1, 1);

    mesh.uv = uv;
}
```

Note that the if the code is executed once in the Start function then the mesh will stay the same throughout the game. However, you can just as easily put the code in the Update function to allow the mesh to be changed each frame (although this will increase the CPU overhead considerably).

Optimizing graphics performance

Leave feedback

Good performance is critical to the success of many games. Below are some simple guidelines for maximizing the speed of your game's **rendering**.

Locate high graphics impact

The graphical parts of your game can primarily impact on two systems of the computer: the GPU and the CPU. The first rule of any optimization is to find **where** the performance problem is, because strategies for optimizing for GPU vs. CPU are quite different (and can even be opposite - for example, it's quite common to make the GPU do more work while optimizing for CPU, and vice versa).

Common bottlenecks and ways to check for them:

GPU is often limited by **fillrate** or memory bandwidth.
Lower the display resolution and run the game. If a lower display resolution makes the game run faster, you may be limited by fillrate on the GPU.
CPU is often limited by the number of batches that need to be rendered.
Check "batches" in the Rendering Statistics window. The more batches are being rendered, the higher the cost to the CPU.
Less-common bottlenecks:

The GPU has too many vertices to process. The number of vertices that is acceptable to ensure good performance depends on the GPU and the complexity of **vertex shaders**. Generally speaking, aim for no more than 100,000 vertices on mobile. A PC manages well even with several million vertices, but it is still good practice to keep this number as low as possible through optimization.
The CPU has too many vertices to process. This could be in skinned meshes, cloth simulation, particles, or other game objects and meshes. As above, it is generally good practice to keep this number as low as possible without compromising game quality. See the section on CPU optimization below for guidance on how to do this.
If rendering is not a problem on the GPU or the CPU, there may be an issue elsewhere - for example, in your script or physics. Use the Unity Profiler to locate the problem.

CPU optimization

To render objects on the screen, the CPU has a lot of processing work to do: working out which lights affect that object, setting up the **shader** and **shader** parameters, and sending drawing commands to the graphics driver, which then prepares the commands to be sent off to the graphics card.

All this "per object" CPU usage is resource-intensive, so if you have lots of visible objects, it can add up. For example, if you have a thousand triangles, it is much easier on the CPU if they are all in one **mesh**, rather than in one **mesh** per triangle (adding up to 1000 meshes). The cost of both scenarios on the GPU is very similar, but the work done by the CPU to render a thousand objects (instead of one) is significantly higher.

Reduce the visible object count. To reduce the amount of work the CPU needs to do:

Combine close objects together, either manually or using Unity's draw call batching.
Use fewer materials in your objects by putting separate textures into a larger texture atlas.
Use fewer things that cause objects to be rendered multiple times (such as reflections, shadows and per-pixel lights).
Combine objects together so that each mesh has at least several hundred triangles and uses only one **Material** for the entire mesh. Note that combining two objects which don't share a material does not give you any performance increase at all. The most common reason for requiring multiple materials is that two meshes don't share the same textures; to optimize CPU performance, ensure that any objects you combine share the same textures.

When using many **pixel** lights in the Forward rendering path, there are situations where combining objects may not make sense. See the Lighting performance section below to learn how to manage this.

GPU: Optimizing model geometry

There are two basic rules for optimizing the geometry of a model:

Don't use any more triangles than necessary
Try to keep the number of UV mapping seams and hard edges (doubled-up vertices) as low as possible
Note that the actual number of vertices that graphics hardware has to process is usually not the same as the number reported by a 3D application. Modeling applications usually display the number of distinct corner points that make up a model (known as the geometric vertex count). For a graphics card, however, some geometric vertices need to be split into two or more logical vertices for rendering purposes. A vertex must be split if it has multiple normals, UV coordinates or vertex colors. Consequently, the vertex count in Unity is usually higher than the count given by the 3D application.

While the amount of geometry in the models is mostly relevant for the GPU, some features in Unity also process models on the CPU (for example, mesh skinning).

Lighting performance

The fastest option is always to create lighting that doesn't need to be computed at all. To do this, use Lightmapping to "bake" static lighting just once, instead of computing it each frame. The process of generating a lightmapped environment takes only a little longer than just placing a light in the **scene** in Unity, **but**:

It runs a lot faster (2–3 times faster for 2-per-pixel lights)
It looks a lot better, as you can bake global illumination and the **lightmapper** can smooth the results
In many cases you can apply simple tricks instead of adding multiple extra lights. For example, instead of adding a light that shines straight into the **camera** to give a **Rim Lighting** effect, add a dedicated `Rim Lighting` computation directly into your shaders (see Surface Shader Examples to learn how to do this).

Lights in forward rendering

Also see: Forward rendering

Per-pixel dynamic lighting adds significant rendering work to every affected pixel, and can lead to objects being rendered in multiple passes. Avoid having more than one **Pixel Light** illuminating any single object on less powerful devices, like mobile or low-end PC GPUs, and use **lightmaps** to light static objects instead of calculating their lighting every frame. Per-vertex dynamic lighting can add significant work to vertex transformations, so try to avoid situations where multiple lights illuminate a single object.

Avoid combining meshes that are far enough apart to be affected by different sets of pixel lights. When you use pixel lighting, each mesh has to be rendered as many times as there are pixel lights illuminating it. If you combine two meshes that are very far apart, it increase the effective size of the combined object. All pixel lights that illuminate any part of this combined object are taken into account during rendering, so the number of rendering passes that need to be made could be increased. Generally, the number of passes that must be made to render the combined object is the **sum of the number of passes for each of the separate objects**, so nothing is gained by combining meshes.

During rendering, Unity finds all lights surrounding a mesh and calculates which of those lights affect it most. The Quality Settings are used to modify how many of the lights end up as pixel lights, and how many as vertex lights. Each light calculates its importance based on how far away it is from the mesh and how intense its illumination is - and some lights are more important than others purely from the game context. For this reason, every light has a **Render Mode** setting which can be set to **Important** or **Not Important**; lights marked as **Not Important** have a lower rendering overhead.

Example: Consider a driving game in which the player's car is driving in the dark with headlights switched on. The headlights are probably the most visually significant light source in the game, so their **Render Mode** should be set to **Important**. There may be other lights in the game that are less important, like other cars' rear lights or distant lampposts, and which don't improve the visual effect much by being pixel lights. The **Render Mode** for such lights can safely be set to **Not Important** to avoid wasting rendering capacity in places where it has little benefit.

Optimizing per-pixel lighting saves both the CPU and GPU work: the CPU has fewer draw calls to do, and the GPU has fewer vertices to process and pixels to rasterize for all the additional object renders.

GPU: Texture compression and mipmaps

Use Compressed textures to decrease the size of your textures. This can result in faster load times, a smaller memory footprint, and dramatically increased rendering performance. Compressed textures only use a fraction of the memory bandwidth needed for uncompressed 32-bit RGBA textures.

Texture mipmaps

Always enable Generate mipmaps for textures used in a 3D scene. A mipmap texture enables the GPU to use a lower resolution texture for smaller triangles.This is similar to how texture **compression** can help limit the amount of texture data transfered when the GPU is rendering.

The only exception to this rule is when a texel (texture pixel) is known to map 1:1 to the rendered screen pixel, as with **UI** elements or in a 2D game.

LOD and per-layer cull distances

Culling objects involves making objects invisible. This is an effective way to reduce both the CPU and GPU load.

In many games, a quick and effective way to do this without compromising the player experience is to cull small objects more aggressively than large ones. For example, small rocks and debris could be made invisible at long distances, while large buildings would still be visible.

There are a number of ways you can achieve this:

Use the Level Of Detail system

Manually set per-layer culling distances on the camera

Put small objects into a separate layer and set up per-layer cull distances using the Camera.layerCullDistances script function

Realtime shadows

Realtime shadows are nice, but they can have a high impact on performance, both in terms of extra draw calls for the CPU and extra processing on the GPU. For further details, see the Light Performance page.

GPU: Tips for writing high-performance shaders

Different platforms have vastly different performance capabilities; a high-end PC GPU can handle much more in terms of graphics and shaders than a low-end mobile GPU. The same is true even on a single platform; a fast GPU is dozens of times faster than a slow integrated GPU.

GPU performance on mobile platforms and low-end PCs is likely to be much lower than on your development machine. It's recommended that you manually optimize your shaders to reduce calculations and texture reads, in order to get good performance across low-end GPU machines. For example, some built-in Unity shaders have "mobile" equivalents that are much faster, but have some limitations or approximations.

Below are some guidelines for mobile and low-end PC graphics cards:

Complex mathematical operations

Transcendental mathematical functions (such as `pow`, `exp`, `log`, `cos`, `sin`, `tan`) are quite resource-intensive, so avoid using them where possible. Consider using lookup textures as an alternative to complex math calculations if applicable.

Avoid writing your own operations (such as `normalize`, `dot`, `inversesqrt`). Unity's built-in options ensure that the driver can generate much better code. Remember that the Alpha Test (`discard`) operation often makes your fragment shader slower.

Floating point precision

While the precision (`float` vs `half` vs `fixed`) of floating point variables is largely ignored on desktop GPUs, it is quite important to get a good performance on mobile GPUs. See the Shader Data Types and Precision page for details.

For further details about shader performance, see the Shader Performance page.

Simple checklist to make your game faster

Keep the vertex count below 200K and 3M per frame when building for PC (depending on the target GPU).
If you're using built-in shaders, pick ones from the **Mobile** or **Unlit** categories. They work on non-mobile platforms as well, but are simplified and approximated versions of the more complex shaders.
Keep the number of different materials per scene low, and share as many materials between different objects as possible.
Set the `Static` property on a non-moving object to allow internal optimizations like static batching.
Only have a single (preferably directional) `pixel light` affecting your geometry, rather than multiples.
Bake lighting rather than using dynamic lighting.
Use compressed texture formats when possible, and use 16-bit textures over 32-bit textures.
Avoid using fog where possible.
Use Occlusion Culling to reduce the amount of visible geometry and draw-calls in cases of complex static scenes with lots of occlusion. Design your levels with occlusion culling in mind.
Use skyboxes to "fake" distant geometry.
Use pixel shaders or texture combiners to mix several textures instead of a multi-pass approach.
Use `half` precision variables where possible.
Minimize use of complex mathematical operations such as `pow`, `sin` and `cos` in pixel shaders.
Use fewer textures per fragment.

See Also

Unity Profiler Window.
Light Performance.

Draw call batching

To draw a **GameObject** on the screen, the engine has to issue a draw call to the graphics API (such as OpenGL or Direct3D). Draw calls are often resource-intensive, with the graphics API doing significant work for every draw call, causing performance overhead on the CPU side. This is mostly caused by the state changes done between the draw calls (such as switching to a different Material), which causes resource-intensive validation and translation steps in the graphics driver.

Unity uses two techniques to address this:

Dynamic batching: for small enough Meshes, this transforms their vertices on the CPU, groups many similar vertices together, and draws them all in one go.
Static batching: combines static (not moving) GameObjects into big Meshes, and renders them in a faster way.
Built-in batching has several benefits compared to manually merging GameObjects together; most notably, GameObjects can still be culled individually. However, it also has some downsides; static batching incurs memory and storage overhead, and dynamic batching incurs some CPU overhead.

Material set-up for batching

Only GameObjects sharing the same Material can be batched together. Therefore, if you want to achieve good batching, you should aim to share Materials among as many different GameObjects as possible.

If you have two identical Materials which differ only in Texture, you can combine those Textures into a single big Texture. This process is often called Texture atlasing (see the Wikipedia page on Texture atlases for more information). Once Textures are in the same atlas, you can use a single Material instead.

If you need to access shared Material properties from the **scripts**, then it is important to note that modifying Renderer.material creates a copy of the Material. Instead, use Renderer.sharedMaterial to keep Materials shared.

Shadow casters can often be batched together while **rendering**, even if their Materials are different. Shadow casters in Unity can use dynamic batching even with different Materials, as long as the values in the Materials needed by the shadow pass are the same. For example, many crates could use Materials with different Textures on them, but for the shadow caster **rendering** the textures are not relevant, so in this case they can be batched together.

Dynamic batching (Meshes)

Unity can automatically batch moving GameObjects into the same draw call if they share the same Material and fulfill other criteria. Dynamic batching is done automatically and does not require any additional effort on your side.

Batching dynamic GameObjects has certain overhead per vertex, so batching is applied only to Meshes containing no more than 900 vertex attributes, and no more than 300 vertices.
If your **Shader** is using Vertex Position, Normal and single UV, then you can batch up to 300 verts, while if your **Shader** is using Vertex Position, Normal, UV0, UV1 and Tangent, then only 180 verts.
Note: attribute count limit might be changed in future.
GameObjects are not batched if they contain mirroring on the transform (for example GameObject A with +1 scale and GameObject B with –1 scale cannot be batched together).
Using different Material instances causes GameObjects not to batch together, even if they are essentially the same. The exception is shadow caster rendering.
GameObjects with **lightmaps** have additional renderer parameters: lightmap index and offset/scale into the lightmap. Generally, dynamic lightmapped GameObjects should point to exactly the same lightmap location to be batched.
Multi-pass Shaders break batching.
Almost all Unity Shaders support several Lights in **forward rendering**, effectively doing additional passes for them. The draw calls for “additional per-pixel lights” are not batched.

The Legacy Deferred (light pre-pass) **rendering path** has dynamic batching disabled, because it has to draw GameObjects twice.

Dynamic batching works by transforming all GameObject vertices into world space on the CPU, so it is only an advantage if that work is smaller than doing a draw call. The resource requirements of a draw call depends on many factors, primarily the graphics API used. For example, on consoles or modern APIs like Apple Metal, the draw call overhead is generally much lower, and often dynamic batching cannot be an advantage at all.

Dynamic batching (Particle Systems, Line Renderers, Trail Renderers)

For **components** with geometry that Unity generates dynamically, dynamic batching works differently compared to how it works for Meshes.

For each compatible renderer type, Unity builds all batchable content into 1 large Vertex Buffer.
The renderer sets up the Material state for the batch.
Unity binds the Vertex Buffer to the Graphics Device.
For each Renderer in the batch, Unity updates the offset into the Vertex Buffer, and then submits a new draw call.

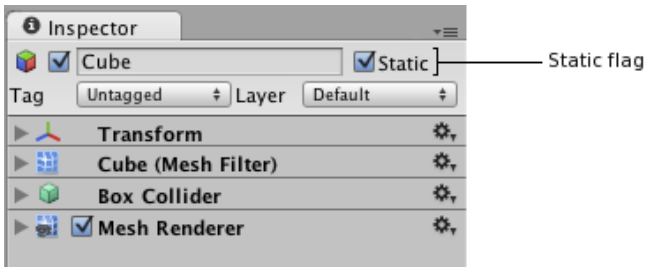
When measuring the cost of the Graphics Device calls, the slowest part of rendering a Component is the set-up of the Material state. Submitting draw calls at different offsets into a shared Vertex Buffer is very fast by comparison.

This approach is very similar to how Unity submits draw calls when using Static batching.

Static batching

Static batching allows the engine to reduce draw calls for geometry of any size provided it shares the same material, and does not move. It is often more efficient than dynamic batching (it does not transform vertices on the CPU), but it uses more memory.

In order to take advantage of static batching, you need to explicitly specify that certain GameObjects are static and do not move, rotate or scale in the game. To do so, mark GameObjects as static using the **Static** checkbox in the **Inspector**:



Using static batching requires additional memory for storing the combined geometry. If several GameObjects shared the same geometry before static batching, then a copy of geometry is created for each GameObject, either in the Editor or at runtime. This might not always be a good idea; sometimes you have to sacrifice rendering performance by avoiding static batching for some GameObjects to keep a smaller memory footprint. For example, marking trees as static in a dense forest level can have serious memory impact.

Internally, static batching works by transforming the static GameObjects into world space and building one shared vertex and index buffer for them. If you have enabled **Optimized** Mesh__ Data__ (in the **Player Settings**) then Unity removes any vertex elements that are not being used by any shader variant when building the vertex buffer. There are some special keyword checks to perform this; for example, if Unity does not detect the LIGHTMAP_ON keyword, it removes lightmap UVs from a batch. Then, for visible GameObjects in the same batch, Unity performs a series of simple draw calls, with almost no state changes in between each one. Technically, Unity does not save API draw calls, but instead saves on state changes between them (which is the resource-intensive part). Batch limits are 64k vertices and 64k indices on most platforms (48k indices on OpenGLES, 32k indices on macOS).

Tips

Currently, only Mesh Renderers, Trail Renderers, Line Renderers, Particle Systems and Sprite Renderers are batched. This means that skinned Meshes, Cloth, and other types of rendering components are not batched.

Renderers only ever batch with other Renderers of the same type.

Semi-transparent Shaders usually require GameObjects to be rendered in back-to-front order for transparency to work. Unity first orders GameObjects in this order, and then tries to batch them, but because the order must be strictly satisfied, this often means less batching can be achieved than with opaque GameObjects.

Manually combining GameObjects that are close to each other can be a very good alternative to draw call batching. For example, a static cupboard with lots of drawers often makes sense to just combine into a single Mesh, either in a 3D modeling application or using Mesh.CombineMeshes.

2017–10–26 Page amended with limited editorial review

Added note on dynamic batching being incompatible with graphics jobs in 2017.2

Modeling characters for optimal performance

Leave feedback

Below are some tips for designing character models to give optimal **rendering** speed.

Use a single skinned Mesh Renderer

You should use only a single skinned Mesh Renderer for each character. Unity optimizes animation using visibility culling and **bounding volume** updates and these optimizations are only activated if you use one Animation component and one skinned **Mesh** Renderer in conjunction. The rendering time for a model could roughly double as a result of using two skinned meshes in place of a single **mesh** and there is seldom any practical advantage in using multiple meshes.

Use as few materials as possible

You should also keep the number of Materials on each mesh as low as possible. The only reason why you might want to have more than one Material on a character is that you need to use different **shaders** for different parts (eg, a special shader for the eyes). However, two or three materials per character should be sufficient in almost all cases.

Use as few bones as possible

A bone hierarchy in a typical desktop game uses somewhere between fifteen and sixty bones. The fewer bones you use, the better the performance will be. You can achieve very good quality on desktop platforms and fairly good quality on mobile platforms with about thirty bones. Ideally, keep the number below thirty for mobile devices, and don't go too far above thirty for desktop games.

Polygon count

The number of polygons you should use depends on the quality you require and the platform you are targeting. For mobile devices, somewhere between 300 and 1500 polygons per mesh will give good results, whereas for desktop platforms the ideal range is about 1500 to 4000. You may need to reduce the polygon count per mesh if the game has lots of characters on screen at any given time.

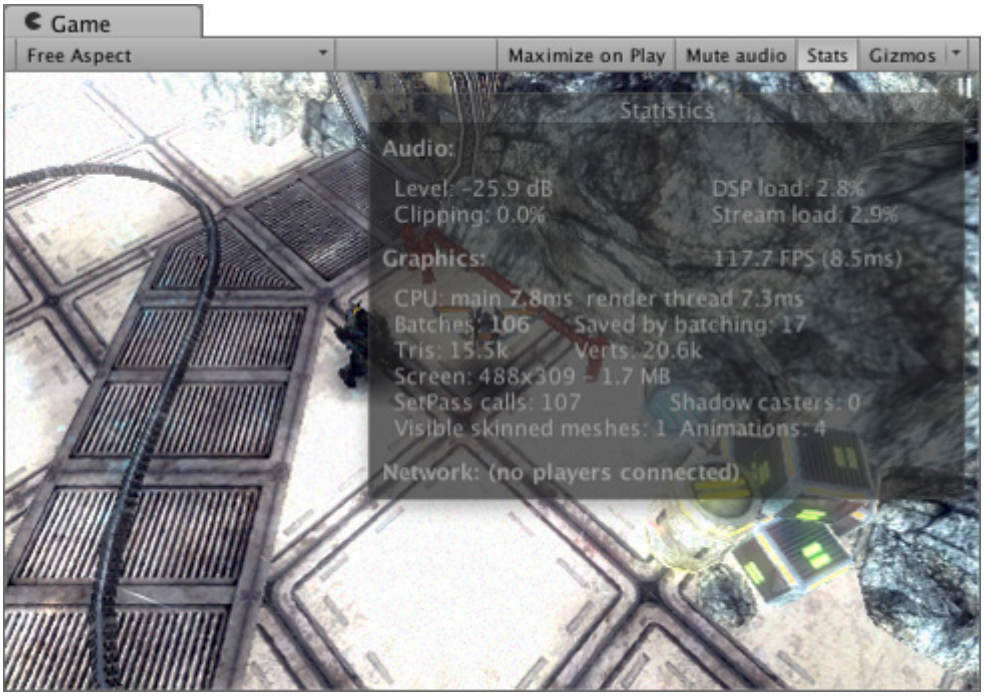
Keep forward and inverse kinematics separate

When animations are imported, a model's inverse kinematic (IK) nodes are baked into forward **kinematics** (FK) and as a result, Unity doesn't need the IK nodes at all. However, if they are left in the model then they will have a CPU overhead even though they don't affect the animation. You can delete the redundant IK nodes in Unity or in the modeling tool, according to your preference. Ideally, you should keep separate IK and FK hierarchies during modeling to make it easier to remove the IK nodes when necessary.

Rendering Statistics Window

Leave feedback

The **Game View** has a **Stats** button in the top right corner. When the button is pressed, an overlay window is displayed which shows realtime **rendering** statistics, which are useful for optimizing performance. The exact statistics displayed vary according to the build target.



Rendering Statistics Window.

The Statistics window contains the following information:-

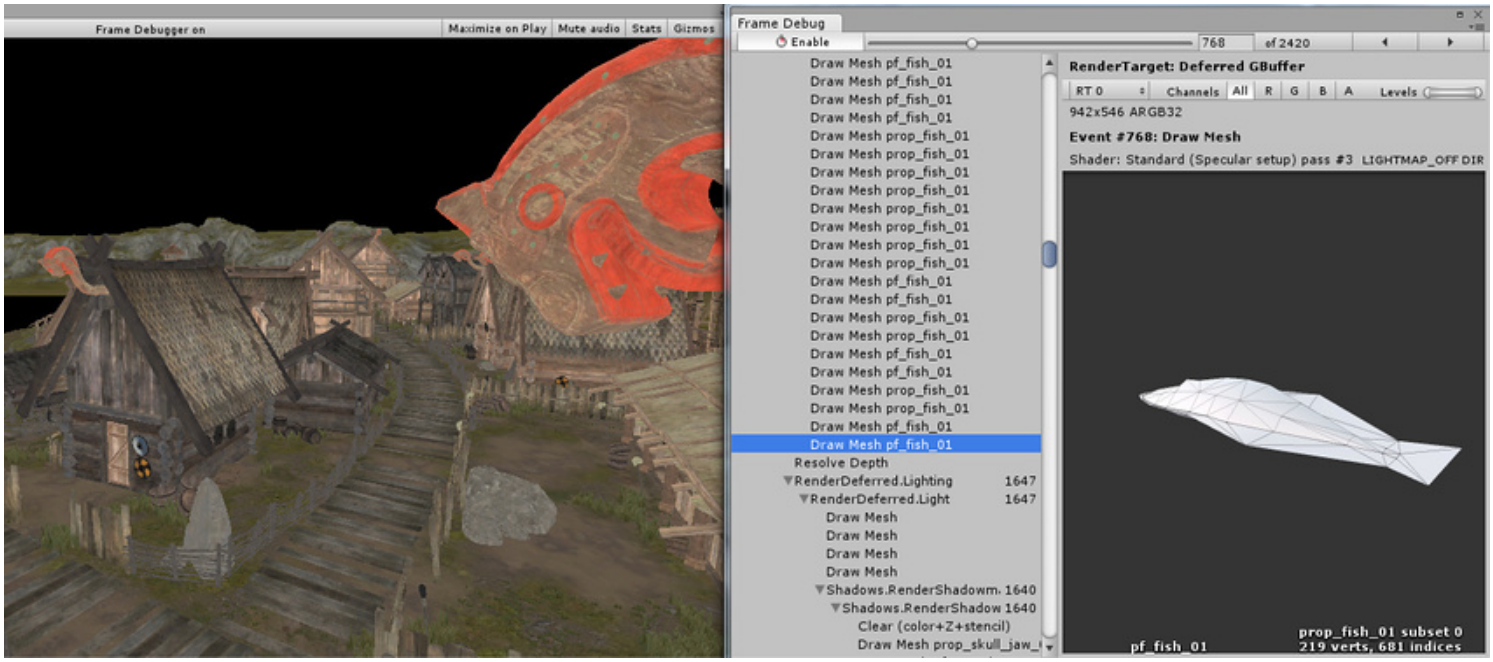
Time per frame and FPS	The amount of time taken to process and render one game frame (and its reciprocal, frames per second). Note that this number only includes the time taken to do the frame update and render the game view; it does not include the time taken in the editor to draw the **scene view**, **inspector** and other editor-only processing.
Batches	"Batching" is where the engine attempts to combine the rendering of multiple objects into a chunk of memory in order to reduce CPU overhead due to resources switching.
Saved by batching	Number of batches that was combined. To ensure good batching, you should share materials between different objects as often as possible. Changing rendering states will break up batches into groups with the same states.
Tris and **Verts**	The number of triangles and vertices drawn. This is mostly important when optimizing for low-end hardware
Screen	The size of the screen, along with its anti-aliasing level and memory usage.
SetPass	The number of rendering passes. Each pass requires Unity runtime to bind a new **shader** which may introduce CPU overhead.
Visible Skinned Meshes	The number of skinned meshes rendered.
Animations	The number of animations playing.

See also the rendering section of the profiler window which provides a more verbose and complete version of these stats.

Frame Debugger

Leave feedback

The **Frame Debugger** lets you freeze playback for a running game on a particular frame and view the individual *draw calls* that are used to render that frame. As well as listing the drawcalls, the debugger also lets you step through them one-by-one so you can see in great detail how the **Scene** is constructed from its graphical elements.



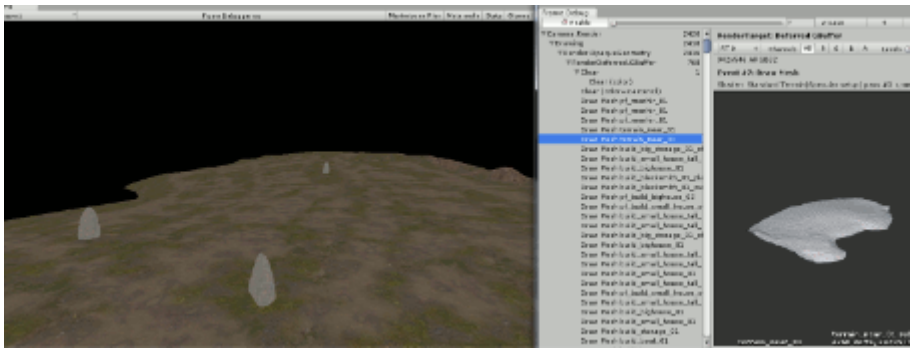
Using the Frame Debugger

The Frame Debugger window (menu: **Window** > **Analysis** > **Frame Debugger**) shows the drawcall information and lets you control the "playback" of the frame under construction.

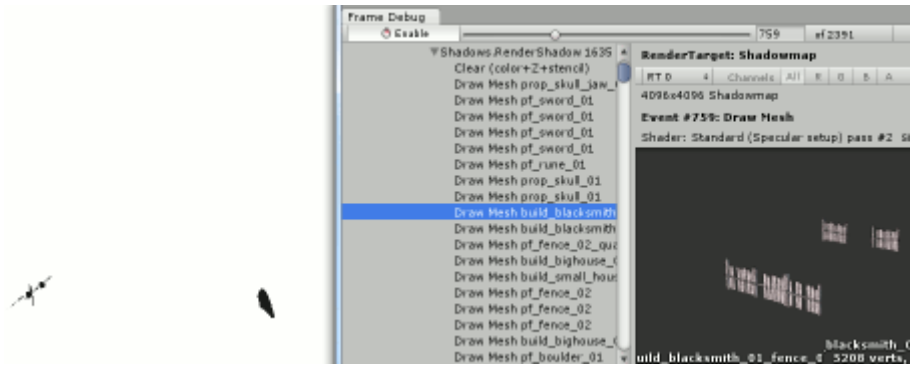
The main list shows the sequence of drawcalls (and other events like framebuffer clear) in the form of a hierarchy that identifies where they originated from. The panel to the right of the list gives further information about the drawcall such as the geometry details and the **shader** used for **rendering**.

Clicking on an item from the list will show the Scene (in the Game view) as it appears up to and including that drawcall. The left and right arrow buttons in the **toolbar** move forward and backward in the list by a single step and you can also use the arrow keys to the same effect. Additionally, the slider at the top of the window lets you "scrub" rapidly through the drawcalls to locate an item of interest quickly. Where a drawcall corresponds to the geometry of a **GameObject**, that object will be highlighted in the main Hierarchy panel to assist identification.

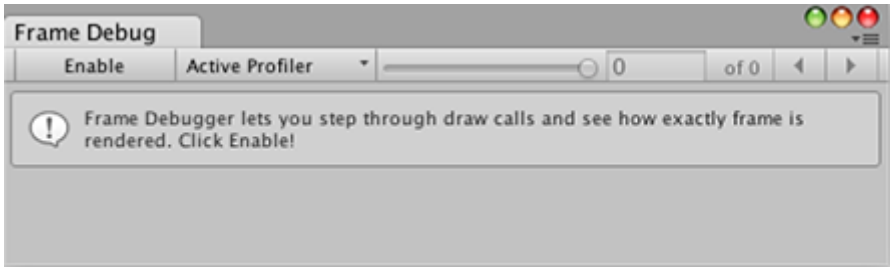
If rendering happens into a RenderTexture at the selected draw call, then contents of that RenderTexture are displayed in the Game view. This is useful for inspecting how various off-screen render targets are built up, for example diffuse g-buffer in **deferred shading**:



Or looking at how the shadow maps are rendered:



Remote Frame Debugger



To use Frame Debugger remotely, the player has to support multithreaded rendering (for ex., WebGL, iOS don't support it, thus frame debugger cannot be run on it), most of the Unity platforms support it, secondly you have to check '**Development Build**' when building.

Note for Desktop platforms: be sure to check 'Run In Background' option before building, otherwise, when you'll connect Frame Debugger to player, it won't reflect any rendering changes until it has focus, assuming you're running both Editor and the player on the same machine, when you'll control Frame Debugger in Editor, you'll take the focus from the player.

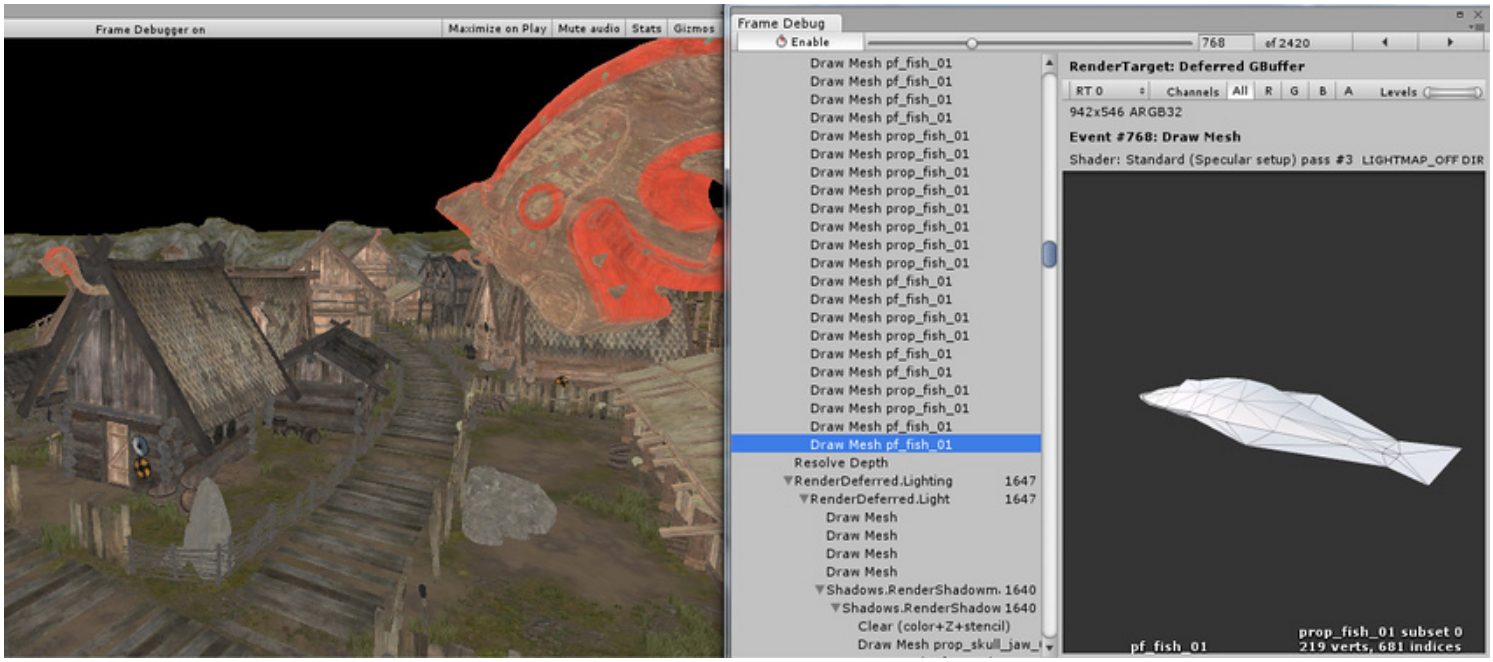
Quick Start:

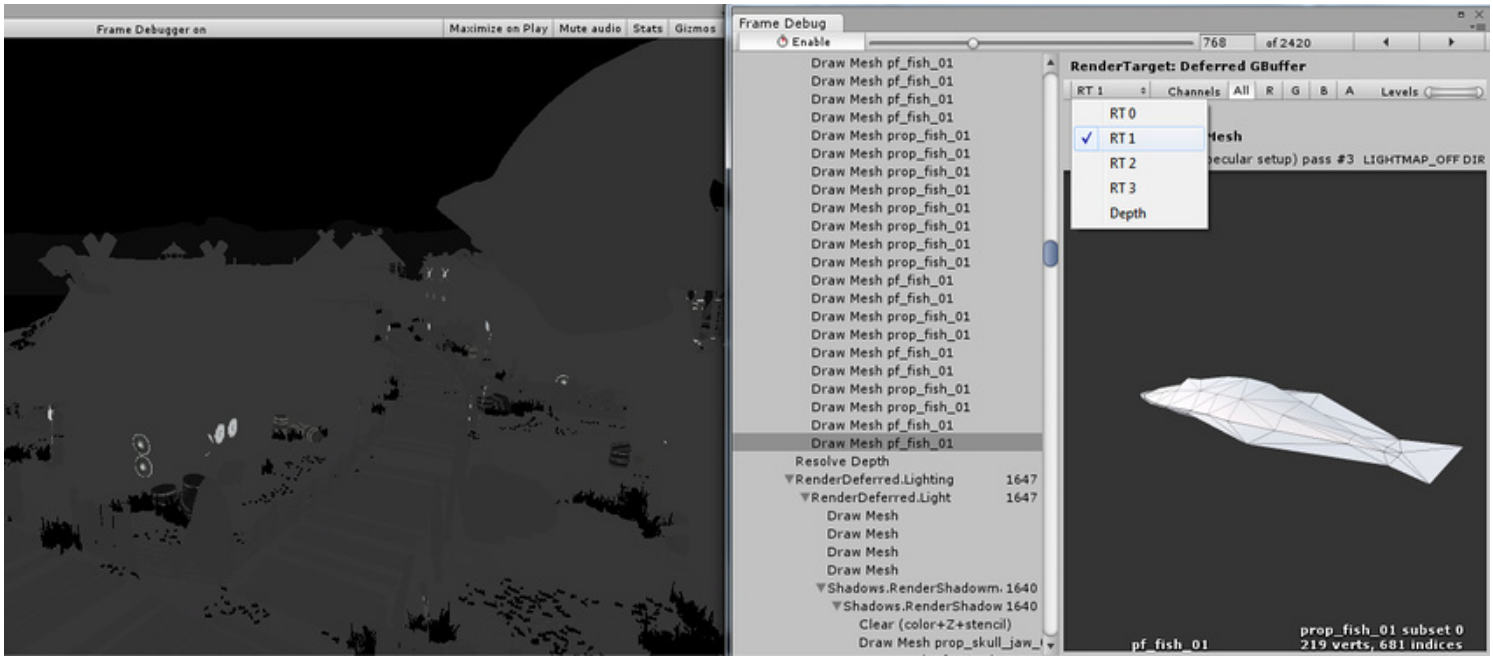
From Editor build the project to target platform (select Development Player)
Run the player
Go back to the Editor
Open Frame Debugger window
Click **Active Profiler** , select the player
Click **Enable**, frame debugger should enable on the player

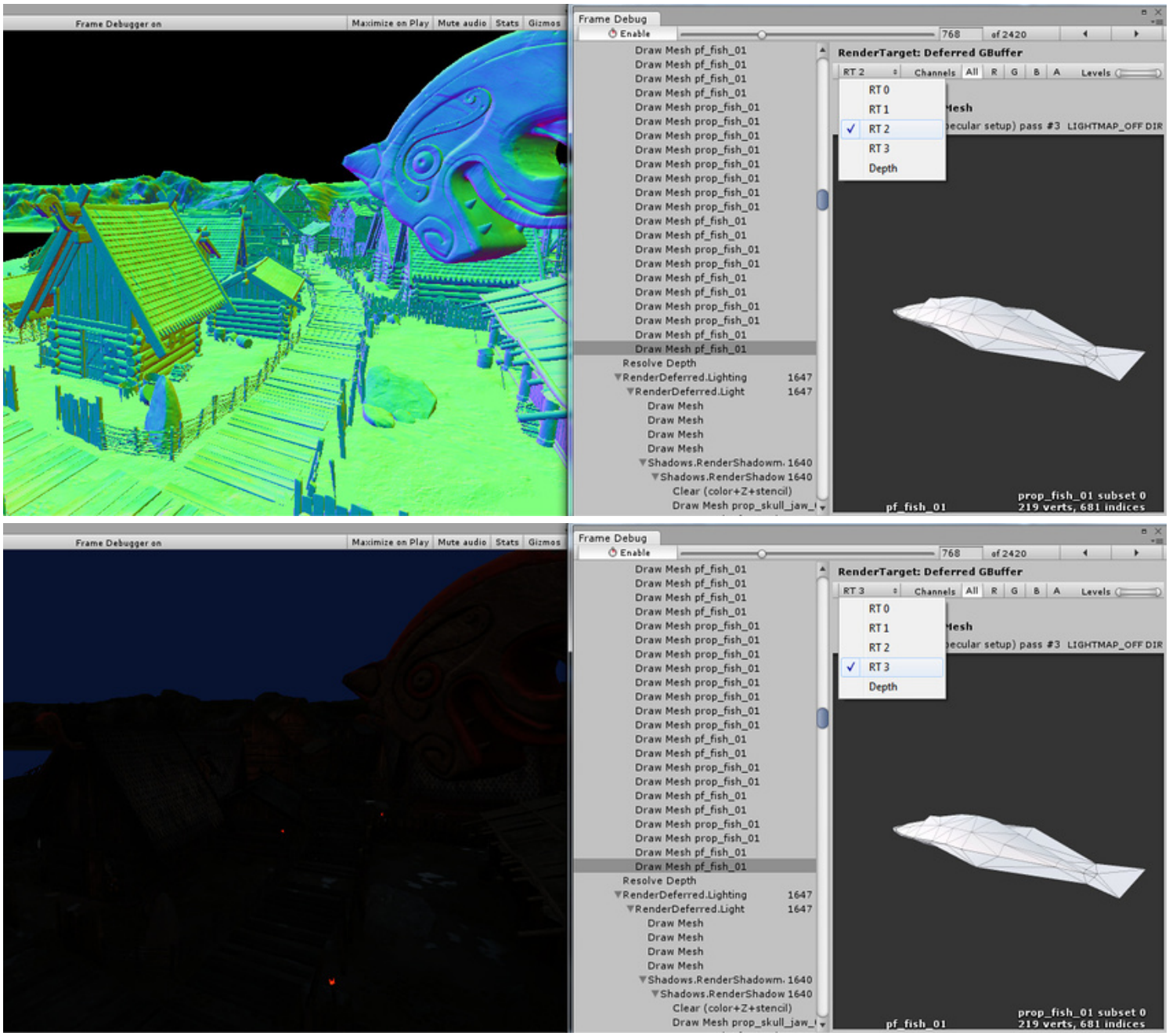
Render target display options

At the top of the information panel is a toolbar which lets you isolate the red, green, blue and alpha channels for the current state of the Game view. Similarly, you can isolate areas of the view according to brightness levels using the *Levels* slider to the right of these channel buttons. These are only enabled when rendering into a RenderTexture.

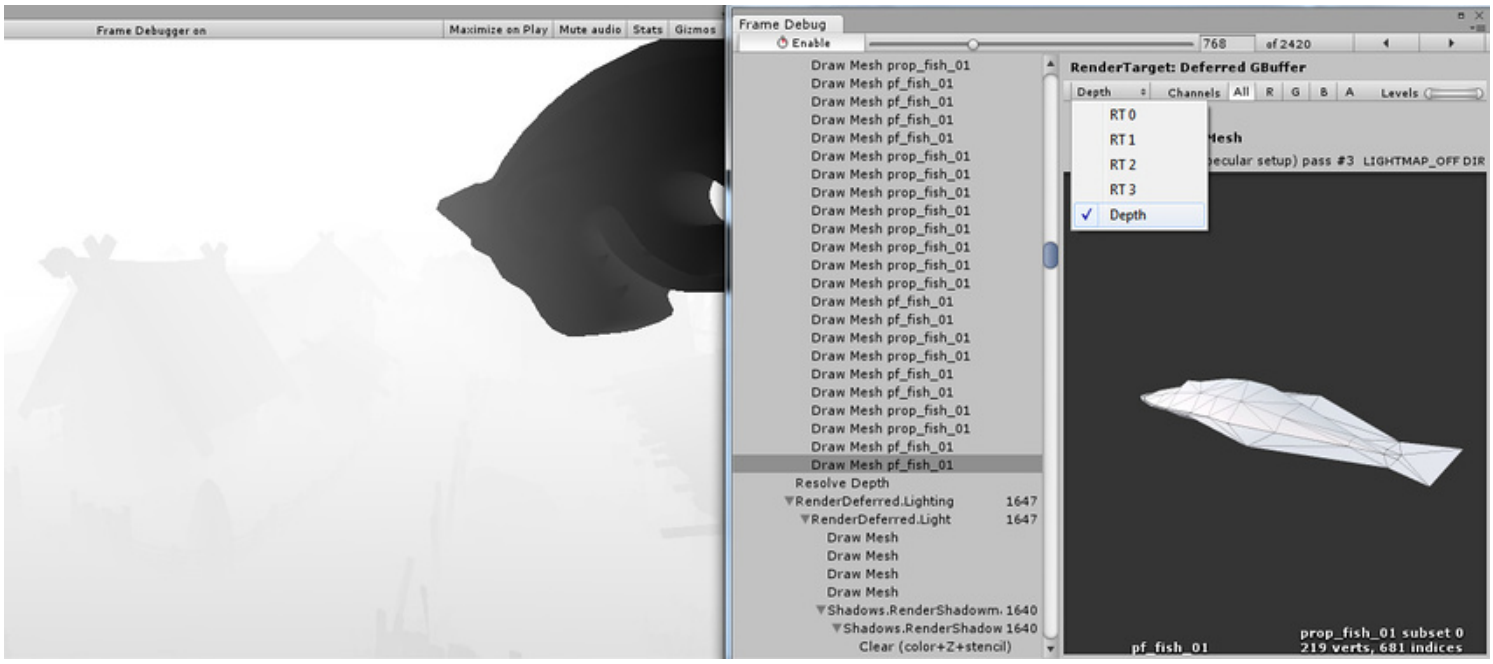
When rendering into multiple render targets at once you can select which one to display in the game view. Shown here are the diffuse, specular, normals and emission/indirect lighting buffers in 5.0 deferred shading mode, respectively:



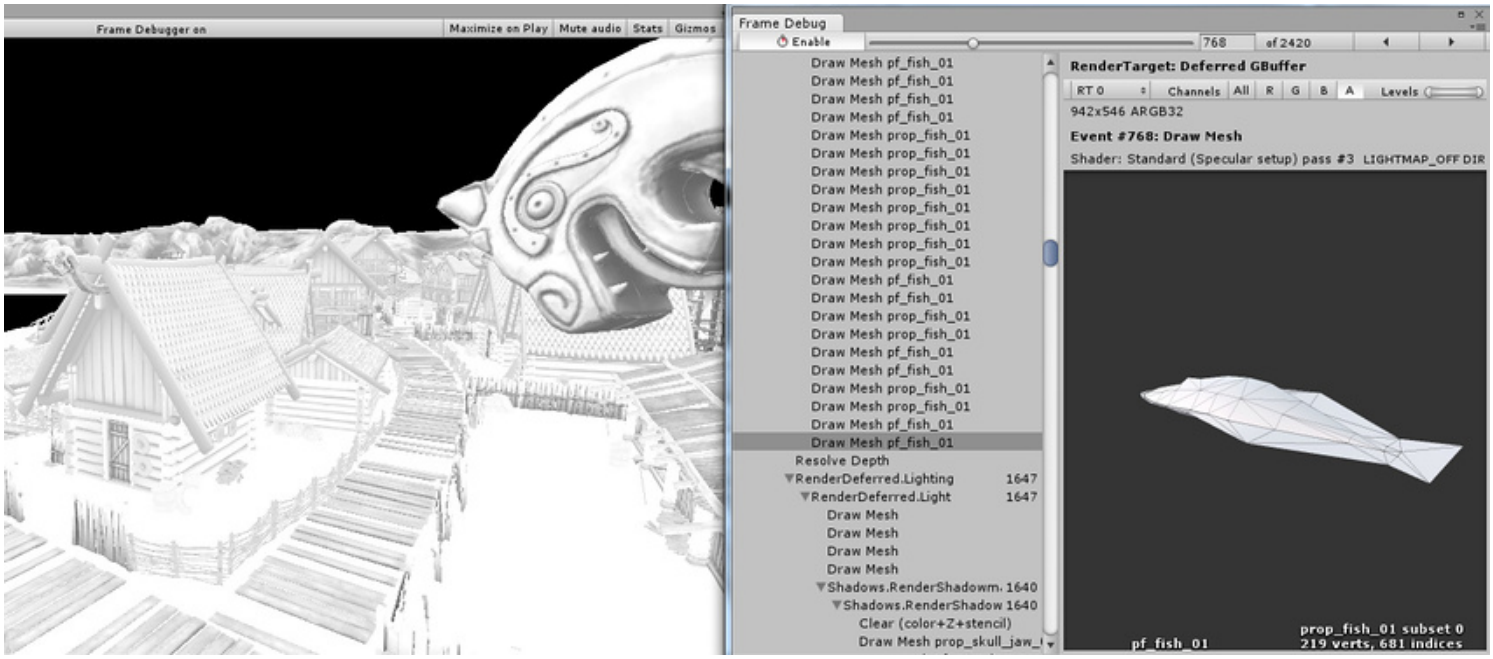


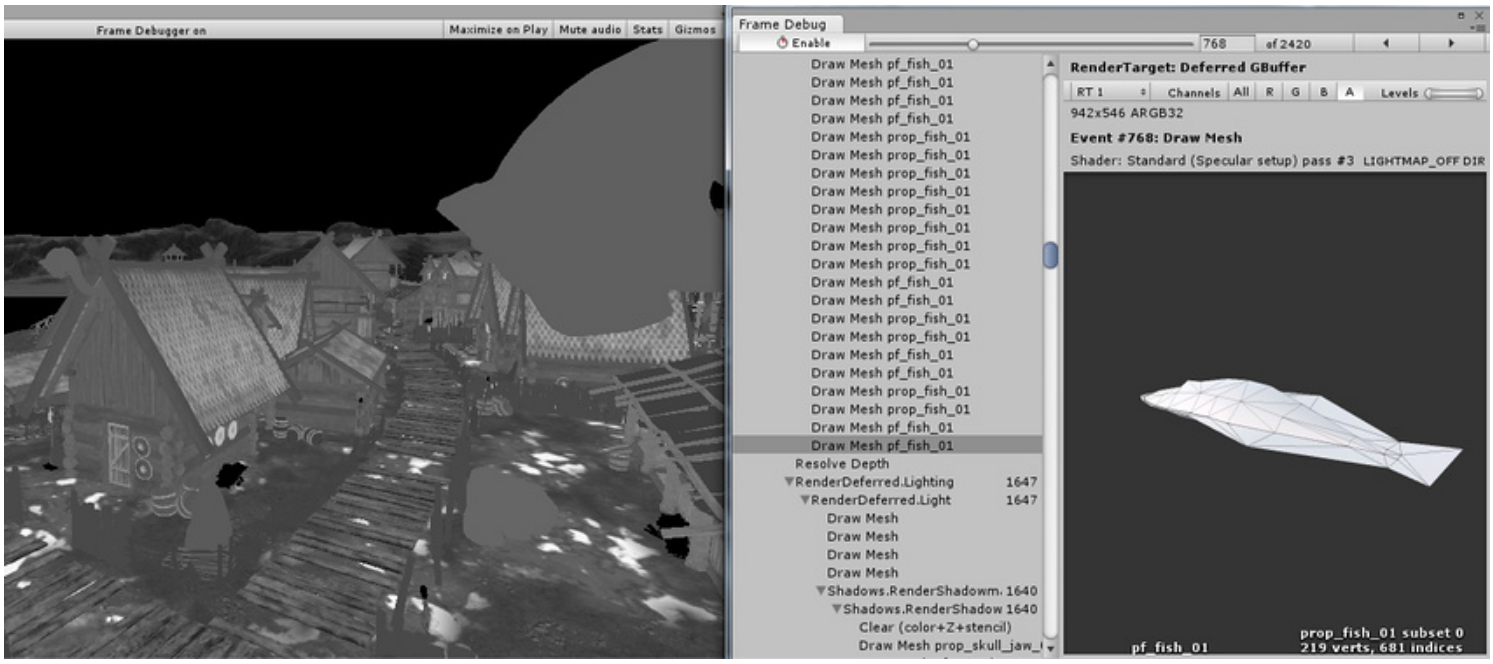


Additionally, you can see the depth buffer contents by picking “Depth” from the dropdown:

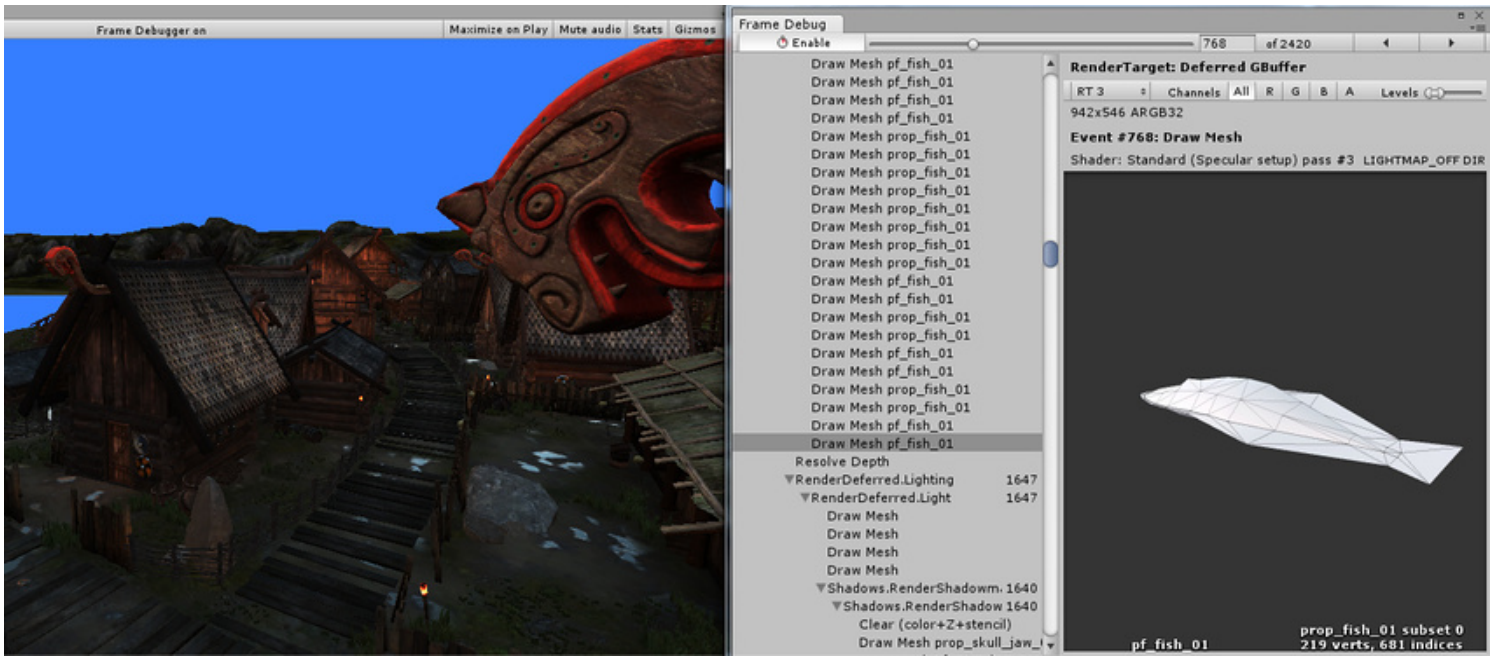


By isolating alpha channel of the **render texture**, you can see occlusion (stored in RT0 alpha) and smoothness (stored in RT1 alpha) of the deferred g-buffer:



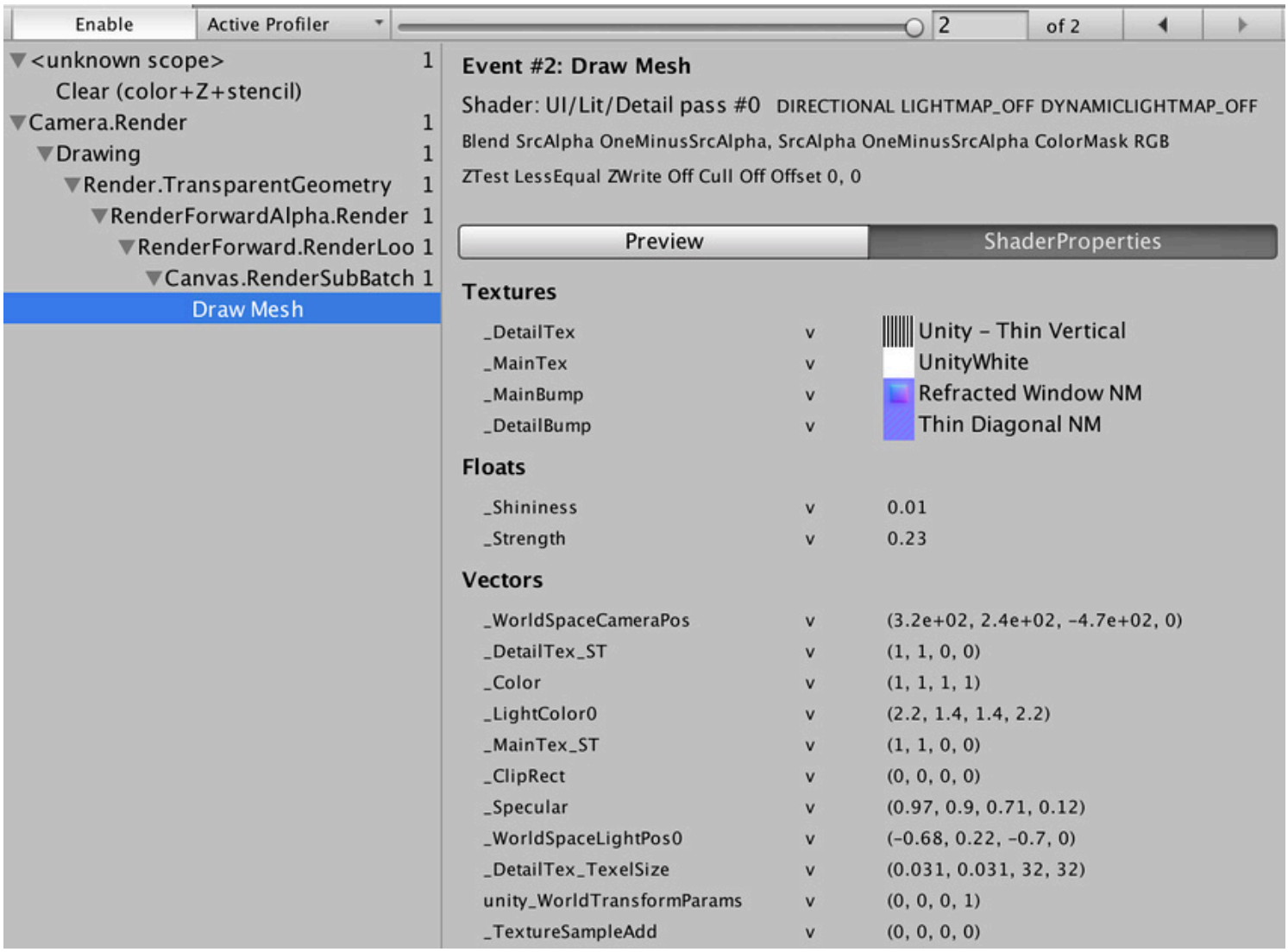


The emission and ambient/indirect lighting in this Scene is very dark; we can make it more visible by changing the Levels slider:



Viewing shader property values

For draw calls, the Frame Debugger can also show shader property values that are used. Click on “Shader Properties” tab to show them:



For each property, the value is shown, as well as which shader stages it was used in (vertex, fragment, geometry, hull, domain). Note that when using OpenGL (e.g. on a Mac), all shader properties are considered to be part of **vertex shader** stage, due to how GLSL shaders work.

In the editor, thumbnails for textures are displayed too, and clicking on them highlights the textures in the project window.

Alternative frame debugging techniques

You could also use external tools to debug rendering. Editor integration exists for easily launching RenderDoc to inspect the Scene or Game view in the Editor.

You can also build a standalone player and run it through any of the following:

Visual Studio graphics debugger
Intel GPA
RenderDoc
NVIDIA NSight
AMD GPU PerfStudio
Xcode GPU Frame Capture
GPU Driver Instruments

When you've done this, capture a frame of rendering, then step through the draw calls and other rendering events to see what's going on. This is a very powerful approach, because these tools can provide you with a lot of information to really drill down.

Optimizing Shader Load Time

Leave feedback

Shaders are small programs that execute on the GPU, and loading them can take some time. Each individual GPU program typically does not take much time to load, but shaders often have a lot of "variants" internally.

For example, the Standard shader, if fully compiled, ends up being many thousands of slightly different GPU programs. This creates two potential problems:

Large numbers of these shader variants increase game build time, and game data size.
Loading large numbers of shader variants during game is slow and takes up memory.

Shader build time stripping

While building the game, Unity can detect that some of the internal shader variants are not used by the game, and skip them from build data. Build-time stripping is done for:

Individual shader features, for shaders that use `#pragma shader_feature`. If none of the used materials use a particular variant, then it is not included into the build. See internal shader variants documentation. Out of built-in shaders, the Standard shader uses this.
Shader variants to handle Fog and Lightmapping modes not used by any of the **scenes** are not included into the game data. See Graphics Settings if you want to override this behavior.
Combination of the above often substantially cuts down on shader data size. For example, a fully compiled Standard shader would take several hundred megabytes, but in typical projects it often ends up taking just a couple megabytes (and is often compressed further by the application packaging process).

Default Unity shader loading behavior

Under all default settings, Unity loads the shaderlab Shader object into memory, but does not create the internal shader variants until they are actually needed.

This means that shader variants that are included into the game build can still potentially be used, but there's no memory or load time cost paid until they are needed. For example, shaders always include a variant to handle point lights with shadows, but if you never end up using a point light with shadows in your game, then there's no point in loading this particular variant.

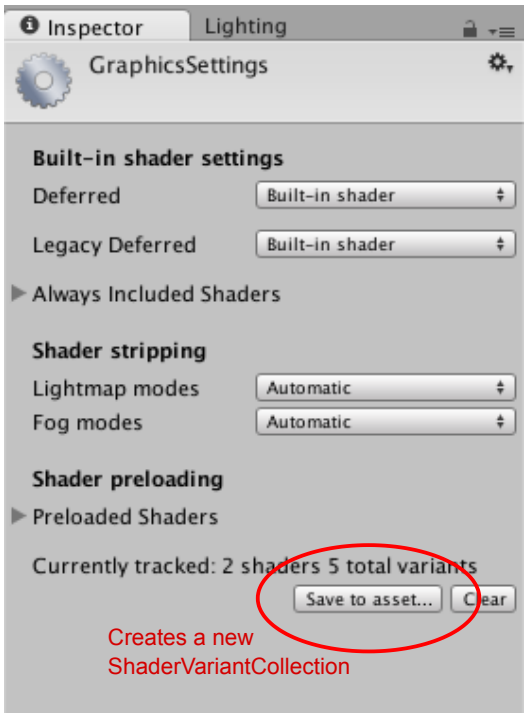
One downside of this default behavior, however, is a possible hiccup for when some shader variant is needed for the first time - since a new GPU program code has to be loaded into the graphics driver. This is often undesirable during gameplay, so Unity has **ShaderVariantCollection assets** to help solve that.

Shader Variant Collections

ShaderVariantCollection is an asset that is basically a list of Shaders, and for each of them, a list of Pass types and shader keyword combinations to load.

Shader variant collection inspector

To help with creating these assets based on actually used shaders and their variants, the editor can track which shaders and their variants are actually used. In Graphics Settings, there is a button to create a new ShaderVariantCollection out of currently tracked shaders, or to clear the currently tracked shader list.



Creating ShaderVariantCollection from shaders used by editor

Once you have some ShaderVariantCollection assets, you can set for these variants to be automatically preloaded while loading the application (under Preloaded Shaders list in Graphics Settings), or you can preload an individual shader variant collection from a script.

The Preloaded Shaders list is intended for frequently used shaders. Shader variants that are listed there the are loaded into memory for entire lifetime of the application. This may use significant amount of memory for ShaderVariantCollections assets that include large number of variants. To avoid that, ShaderVariantCollection assets should be created at smaller granularity and loaded from a script. One strategy is to record used shader variants for each scene, save them into separate ShaderVariantCollections assets and load them on scene startup.

See ShaderVariantCollection scripting class.

See Also

Optimizing Graphics Performance.
Graphics Settings.
Shaders reference.

Layers

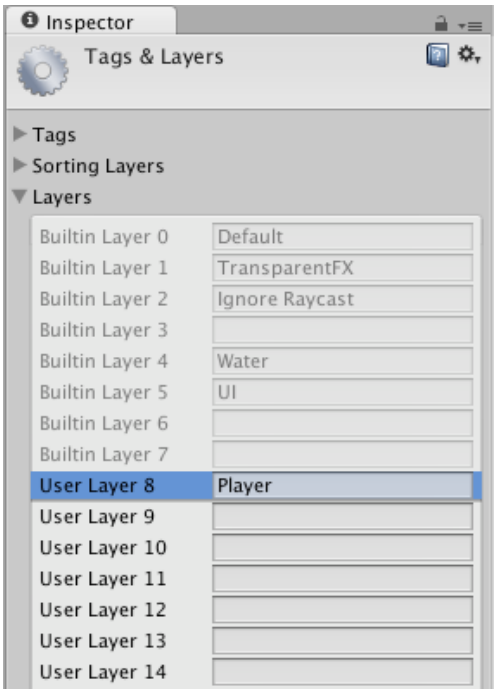
Leave feedback

Layers are most commonly used by **Cameras** to render only a part of the **scene**, and by **Lights** to illuminate only parts of the **scene**. But they can also be used by raycasting to selectively ignore **colliders** or to create collisions.

Creating Layers

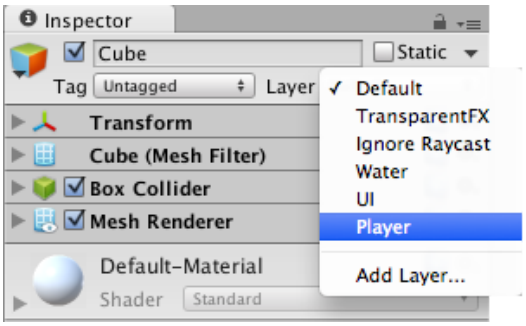
The first step is to create a new layer, which we can then assign to a **GameObject**. To create a new layer, open the Edit menu and select **Project Settings->Tags and Layers**.

We create a new layer in one of the empty User Layers. We choose layer 8.



Assigning Layers

Now that you have created a new layer, you have to assign the layer to one of the game objects.

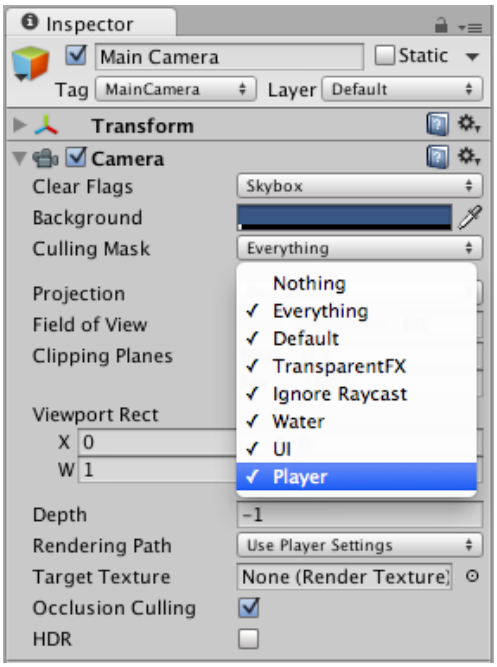


In the tag manager we assigned the Player layer to be in layer 8.

Drawing only a part of the scene with the camera's culling mask

Using the camera's **culling mask**, you can selectively render objects which are in one particular layer. To do this, select the camera that should selectively render objects.

Modify the culling mask by checking or unchecking layers in the culling mask property.



Be aware that **UI** elements aren't culled. Screen space canvas children do not respect the camera's culling mask.

Casting Rays Selectively

Using layers you can cast rays and ignore colliders in specific layers. For example you might want to cast a ray only against the player layer and ignore all other colliders.

The Physics.Raycast function takes a bitmask, where each bit determines if a layer will be ignored or not. If all bits in the layerMask are on, we will collide against all colliders. If the layerMask = 0, we will never find any collisions with the ray.

```
int layerMask = 1 << 8;

// Does the ray intersect any objects which are in the player layer.
if (Physics.Raycast(transform.position, Vector3.forward, Mathf.Infinity, layerMask))
{
    Debug.Log("The ray hit the player");
}
```

In the real world you want to do the inverse of that however. We want to cast a ray against all colliders except those in the Player layer.

```
void Update ()
{
    // Bit shift the index of the layer (8) to get a bit mask
    int layerMask = 1 << 8;

    // This would cast rays only against colliders in layer 8.
    // But instead we want to collide against everything except layer 8. The ~ operator does
    layerMask = ~layerMask;

    RaycastHit hit;
    // Does the ray intersect any objects excluding the player layer
```

```
    if (Physics.Raycast(transform.position, transform.TransformDirection (Vector3.forward), o
    {
        Debug.DrawRay(transform.position, transform.TransformDirection (Vector3.forward) * hi
        Debug.Log("Did Hit");
    }
    else
    {
        Debug.DrawRay(transform.position, transform.TransformDirection (Vector3.forward) *100
        Debug.Log("Did not Hit");
    }
}
```

When you don't pass a layerMask to the Raycast function, it will only ignore colliders that use the IgnoreRaycast layer. This is the easiest way to ignore some colliders when casting a ray.

Note: Layer 31 is used internally by the Editor's Preview window mechanics. To prevent clashes, do not use this layer.

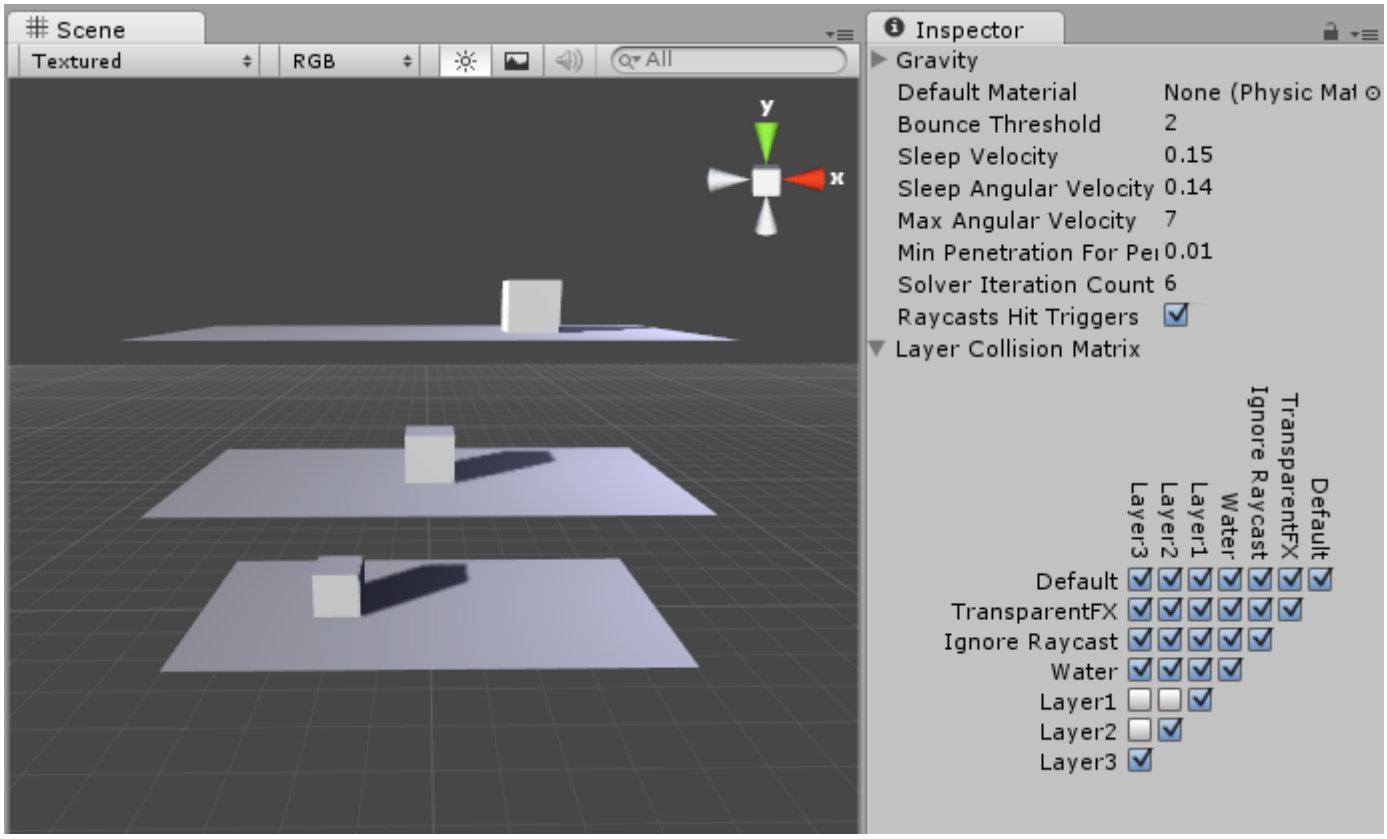
2017–05–08 Page amended with limited editorial review

Culling mask information updated in Unity 2017.1

Layer-based collision detection

Leave feedback

Layer-based **collision detection** is a way to make a **GameObject** collide with another **GameObject** that is set up to a specific Layer or Layers.



Objects colliding with their own layer

The image above shows six GameObjects (3 planes, 3 cubes) in the **Scene view**, and the **Layer** Collision__A collision occurs when the physics engine detects that the colliders of two GameObjects make contact or overlap, when at least one has a rigidbody component and is in motion. More info

See in Glossary Matrix__ in the window to the right. The Layer **Collision** Matrix defines which GameObjects can collide with which Layers.

In the example, the Layer Collision Matrix is set up so that only GameObjects that belong to the same layer can collide:

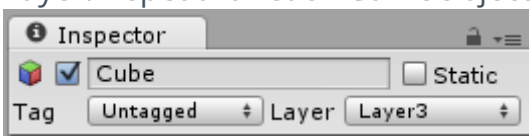
Layer 1 is checked for Layer 1 only
Layer 2 is checked for Layer 2 only
Layer 3 is checked for Layer 3 only

Change this to suit your needs: if, for example, you want Layer 1 to collide with Layer 2 and 3, but not with Layer 1, find the row for **Layer 1**, then check the boxes for the **Layer 2** and **Layer 3** colums, and leave the **Layer 1** column checkbox blank.

Setting up layer-based collision detection

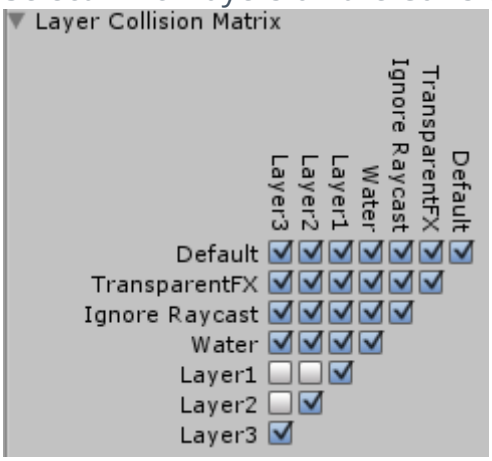
To select a Layer for your GameObjects to belong to, select the GameObject, navigate to the **Inspector** window, select the **Layer** dropdown at the top, and either choose a Layer or add a new

Layer. Repeat for each GameObject until you have finished assigning your GameObjects to Layers.



In the Unity menu bar, go to **Edit** > **Project Settings** > **Physics** to open the **Physics Manager** window.

Select which layers on the Collision Matrix will interact with the other layers by checking them.



Graphics Reference

Leave feedback

This section goes into more depth about Unity's graphical features.

Cameras Reference

Leave feedback

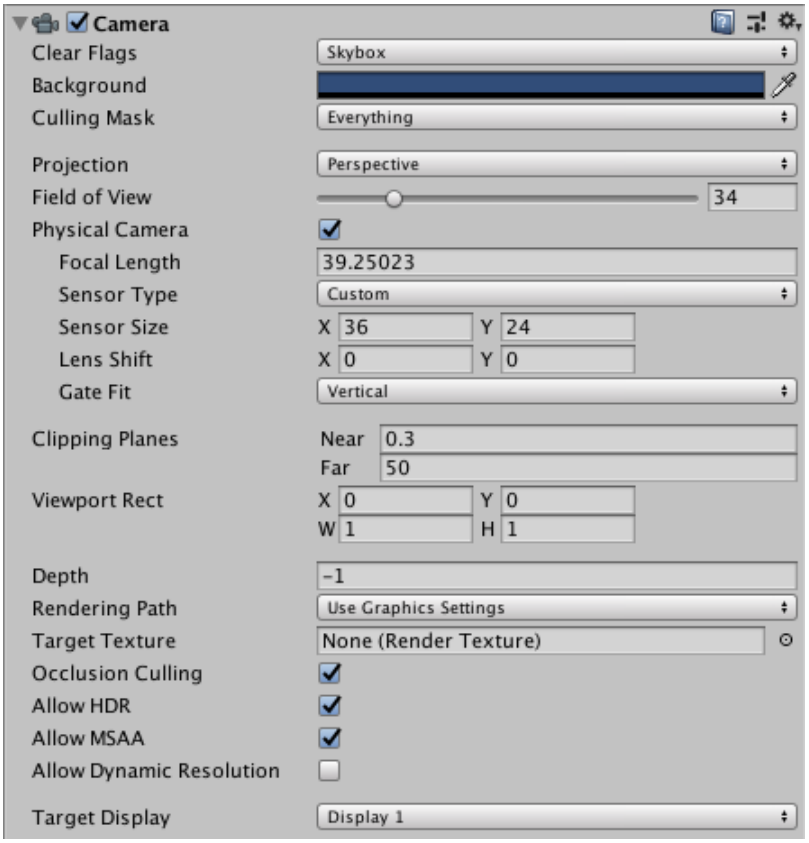
This section contains more detailed information on **Cameras**.

Camera

Leave feedback

SWITCH TO SCRIPTING

Cameras are the devices that capture and display the world to the player. By customizing and manipulating cameras, you can make the presentation of your game truly unique. You can have an unlimited number of cameras in a **scene**. They can be set to render in any order, at any place on the screen, or only certain parts of the screen.



Properties

Property:	*Function:*
Clear Flags	Determines which parts of the screen will be cleared. This is handy when using multiple Cameras to draw different game elements.
Background	The color applied to the remaining screen after all elements in view have been drawn and there is no **skybox**.
Culling Mask	Includes or omits layers of objects to be rendered by the Camera. Assigns layers to your objects in the **Inspector**.
Projection	Toggles the camera's capability to simulate perspective.
Perspective	Camera will render objects with perspective intact.
Orthographic	Camera will render objects uniformly, with no sense of perspective. **NOTE:** Deferred rendering is not supported in Orthographic mode. **Forward rendering** is always used.
Size (when Orthographic is selected)	The **viewport** size of the Camera when set to Orthographic.
Field of view (when Perspective is selected)	The width of the Camera's view angle, measured in degrees along the local Y axis.

Property:	*Function:*
Clipping Planes	Distances from the camera to start and stop rendering.
Near	The closest point relative to the camera that drawing will occur.
Far	The furthest point relative to the camera that drawing will occur.
Viewport Rect	Four values that indicate where on the screen this camera view will be drawn. Measured in Viewport Coordinates (values 0–1).
X	The beginning horizontal position that the camera view will be drawn.
Y	The beginning vertical position that the camera view will be drawn.
W (Width)	Width of the camera output on the screen.
H (Height)	Height of the camera output on the screen.
Depth	The camera's position in the draw order. Cameras with a larger value will be drawn on top of cameras with a smaller value.
Rendering Path	Options for defining what rendering methods will be used by the camera.
Use Player Settings	This camera will use whichever Rendering Path is set in the Player Settings.
Vertex Lit	All objects rendered by this camera will be rendered as Vertex-Lit objects.
Forward	All objects will be rendered with one pass per material.
Deferred Lighting	All objects will be drawn once without lighting, then lighting of all objects will be rendered together at the end of the render queue. **NOTE:** *If the camera's projection mode is set to Orthographic, this value is overridden, and the camera will always use Forward rendering.*
Target Texture	Reference to a Render Texture that will contain the output of the Camera view. Setting this reference will disable this Camera's capability to render to the screen.
HDR	Enables High Dynamic Range rendering for this camera.
Target Display	Defines which external device to render to. Between 1 and 8.

Details

Cameras are essential for displaying your game to the player. They can be customized, scripted, or parented to achieve just about any kind of effect imaginable. For a puzzle game, you might keep the Camera static for a full view of the puzzle. For a first-person shooter, you would parent the Camera to the player character, and place it at the character's eye level. For a racing game, you'd probably have the Camera follow your player's vehicle.

You can create multiple Cameras and assign each one to a different **Depth**. Cameras are drawn from low **Depth** to high **Depth**. In other words, a Camera with a **Depth** of 2 will be drawn on top of a Camera with a depth of 1. You can adjust the values of the **Normalized View Port Rectangle** property to resize and position the Camera's view onscreen. This can create multiple mini-views like missile cams, map views, rear-view mirrors, **etc**.

Render path

Unity supports different rendering paths. You should choose which one you use depending on your game content and target platform / hardware. Different rendering paths have different features and performance characteristics that mostly affect lights and shadows. The rendering path used by your project is chosen in **Player Settings**. Additionally, you can override it for each Camera.

For more information on rendering paths, check the rendering paths page.

Clear Flags

Each Camera stores color and depth information when it renders its view. The portions of the screen that are not drawn in are empty, and will display the skybox by default. When you are using multiple Cameras, each one stores its own color and depth information in buffers, accumulating more data as each Camera renders. As any particular Camera in your scene renders its view, you can set the **Clear Flags** to clear different collections of the buffer information. To do this, choose one of the following four options:

Skybox

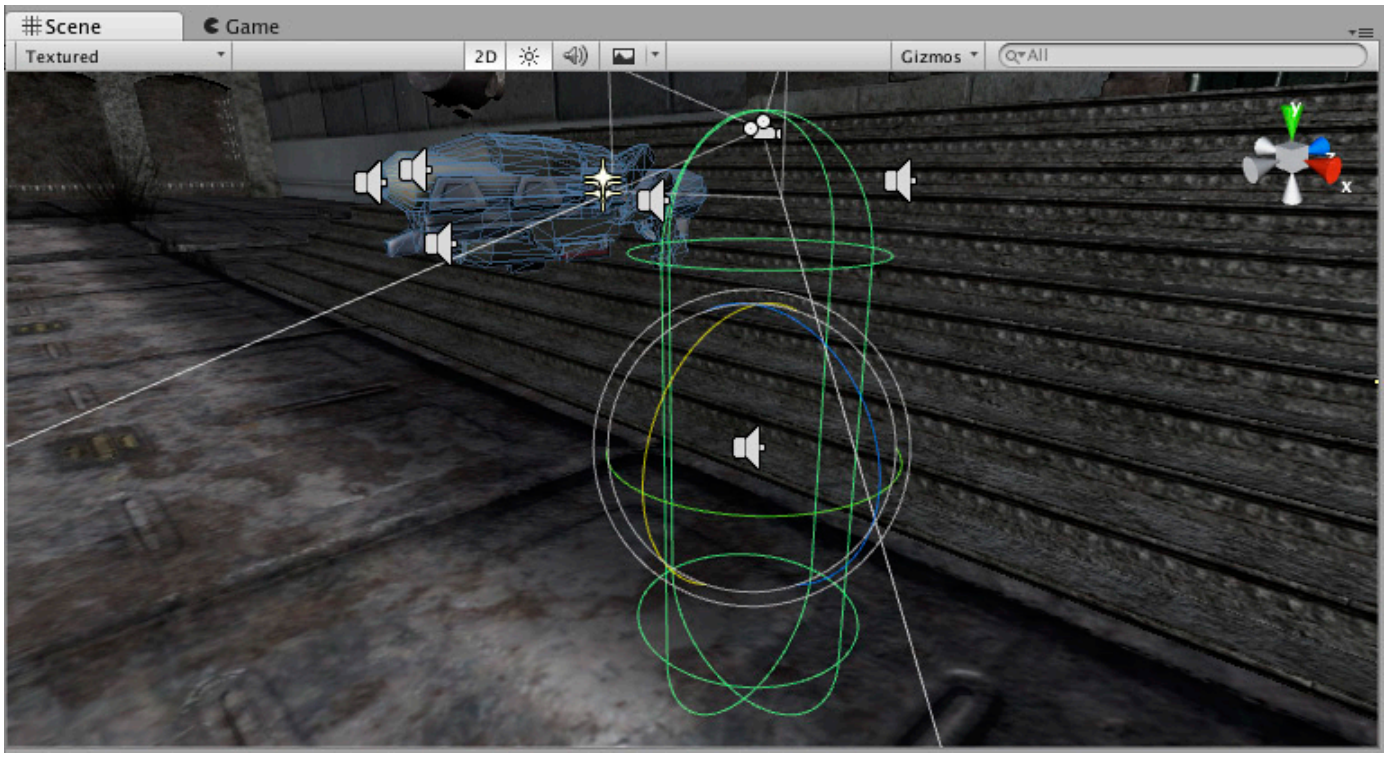
This is the default setting. Any empty portions of the screen will display the current Camera's skybox. If the current Camera has no skybox set, it will default to the skybox chosen in the Lighting Window (menu: **Window** > **Rendering** > **Lighting Settings**). It will then fall back to the **Background Color**. Alternatively a Skybox component can be added to the camera. If you want to create a new Skybox, you can use this guide.

Solid color

Any empty portions of the screen will display the current Camera's **Background Color**.

Depth only

If you want to draw a player's gun without letting it get clipped inside the environment, set one Camera at **Depth** 0 to draw the environment, and another Camera at **Depth** 1 to draw the weapon alone. Set the weapon Camera's **Clear Flags** to **depth only**. This will keep the graphical display of the environment on the screen, but discard all information about where each object exists in 3-D space. When the gun is drawn, the opaque parts will completely cover anything drawn, regardless of how close the gun is to the wall.



The gun is drawn last, after clearing the depth buffer of the cameras before it

Don't clear

This mode does not clear either the color or the depth buffer. The result is that each frame is drawn over the next, resulting in a smear-looking effect. This isn't typically used in games, and would more likely be used with a custom **shader**.

Note that on some GPUs (mostly mobile GPUs), not clearing the screen might result in the contents of it being undefined in the next frame. On some systems, the screen may contain the previous frame image, a solid black screen, or random colored **pixels**.

Clip Planes

The **Near** and **Far Clip Plane** properties determine where the Camera's view begins and ends. The planes are laid out perpendicular to the Camera's direction and are measured from its position. The **Near plane** is the closest location that will be rendered, and the **Far plane** is the furthest.

The clipping planes also determine how depth buffer precision is distributed over the scene. In general, to get better precision you should move the **Near plane** as far as possible.

Note that the near and far clip planes together with the planes defined by the field of view of the camera describe what is popularly known as the camera *frustum*. Unity ensures that when rendering your objects those which are completely outside of this frustum are not displayed. This is called Frustum Culling. Frustum Culling happens irrespective of whether you use **Occlusion Culling** in your game.

For performance reasons, you might want to cull small objects earlier. For example, small rocks and debris could be made invisible at much smaller distance than large buildings. To do that, put small objects into a separate layer and set up per-layer cull distances using Camera.layerCullDistances script function.

Culling Mask

The **Culling Mask** is used for selectively rendering groups of objects using Layers. More information on using layers can be found here.

Normalized Viewport Rectangles

Normalized Viewport Rectangle is specifically for defining a certain portion of the screen that the current camera view will be drawn upon. You can put a map view in the lower-right hand corner of the screen, or a missile-tip view in the upper-left corner. With a bit of design work, you can use **Viewport Rectangle** to create some unique behaviors.

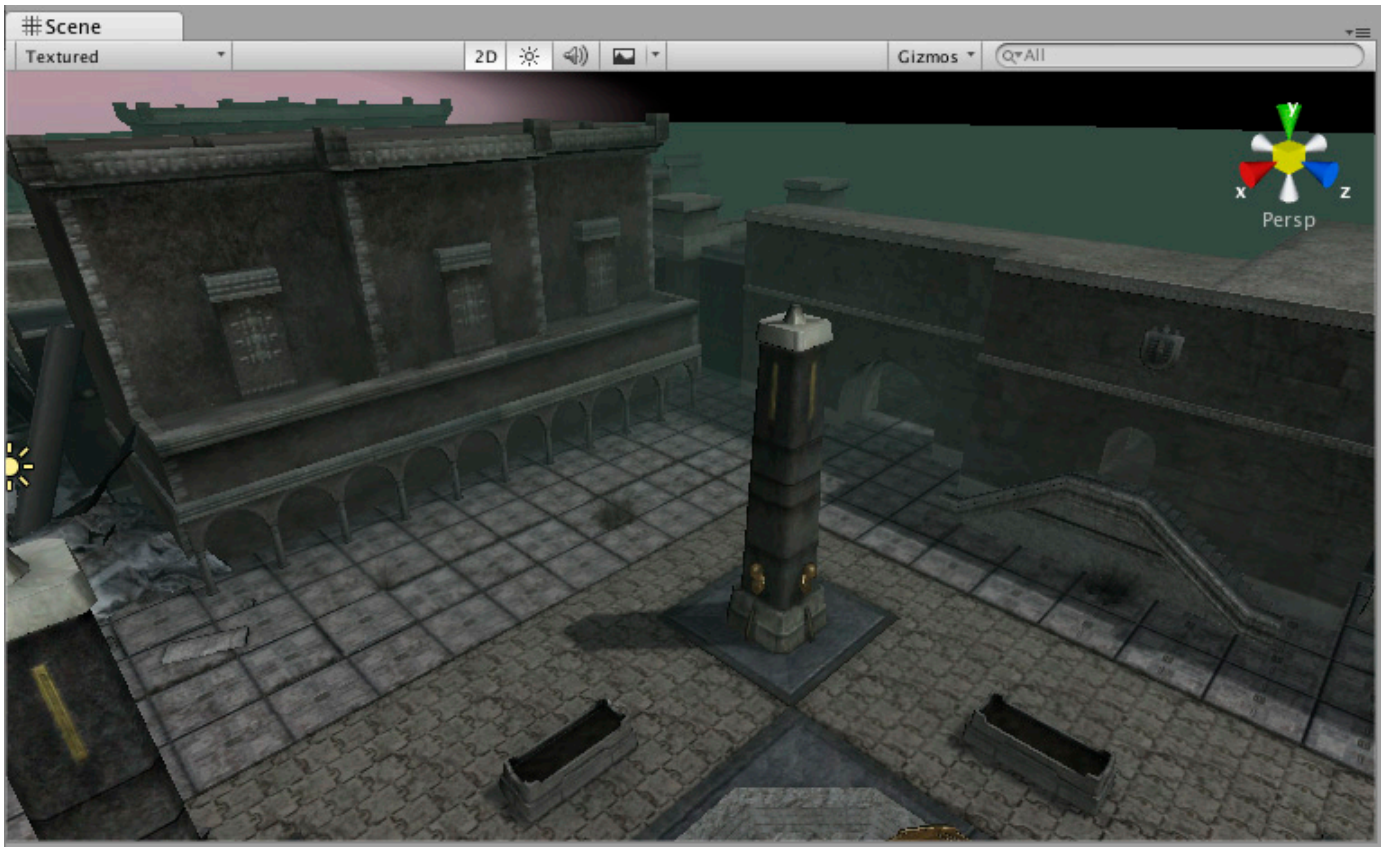
It's easy to create a two-player split screen effect using **Normalized Viewport Rectangle**. After you have created your two cameras, change both camera's H values to be 0.5 then set player one's Y value to 0.5, and player two's Y value to 0. This will make player one's camera display from halfway up the screen to the top, and player two's camera start at the bottom and stop halfway up the screen.

Two-player display created with the Normalized Viewport Rectangle property

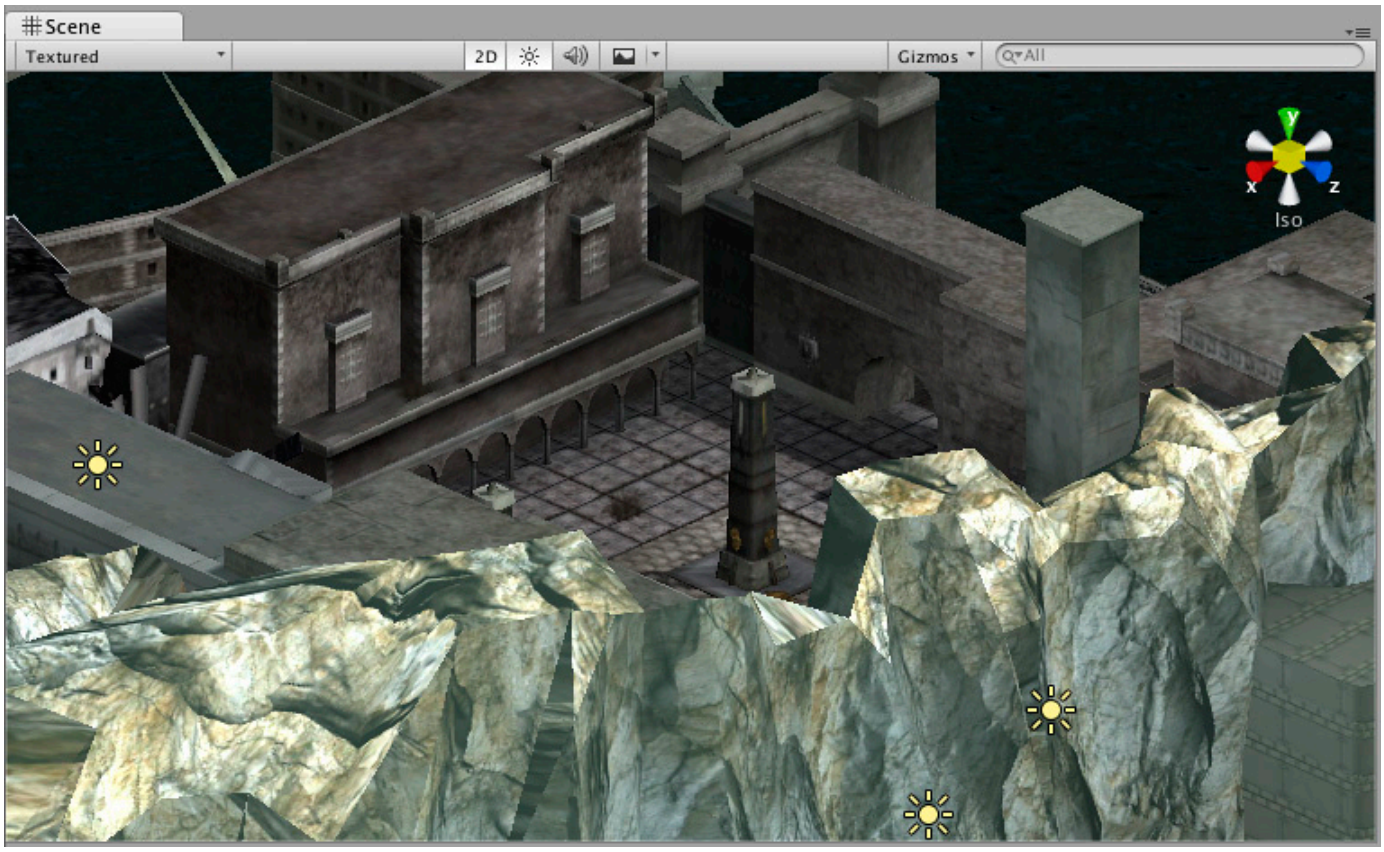
Orthographic

Marking a Camera as **Orthographic** removes all perspective from the Camera's view. This is mostly useful for making isometric or 2D games.

Note that fog is rendered uniformly in orthographic camera mode and may therefore not appear as expected. This is because the Z coordinate of the post-perspective space is used for the fog "depth". This is not strictly accurate for an orthographic camera but it is used for its performance benefits during rendering.



Perspective camera.



Orthographic camera. Objects do not get smaller with distance here!

Render Texture

This will place the camera's view onto a Texture that can then be applied to another object. This makes it easy to create sports arena video monitors, surveillance cameras, reflections etc.

A Render Texture used to create a live arena-cam

Target display

A camera has up to 8 target display settings. The camera can be controlled to render to one of up to 8 monitors. This is supported only on PC, Mac and Linux. In Game View the chosen display in the Camera Inspector will be shown.

Hints

Cameras can be instantiated, parented, and scripted just like any other **GameObject**.
To increase the sense of speed in a racing game, use a high **Field of View**.
Cameras can be used in physics simulation if you add a **Rigidbody Component**.
There is no limit to the number of Cameras you can have in your scenes.
Orthographic cameras are great for making 3D user interfaces.
If you are experiencing depth artifacts (surfaces close to each other flickering), try setting **Near Plane** to as large as possible.
Cameras cannot render to the Game Screen and a Render Texture at the same time, only one or the other.
There's an option of rendering a Camera's view to a texture, called Render-to-Texture, for even more interesting effects.
Unity comes with pre-installed Camera scripts, found in **Components > Camera Control**. Experiment with them to get a taste of what's possible.

Flare Layer

Leave feedback

SWITCH TO SCRIPTING

The **Flare Layer Component** can be attached to Cameras to make Lens Flares appear in the image. By default, Cameras have a Flare Layer already attached.

GUI Layer (Legacy)

Leave feedback

SWITCH TO SCRIPTING

Please Note: *This component relates to legacy methods for drawing UI textures and images to the screen. You should use Unity's up-to-date UI system instead. This is also unrelated to the IMGUI system.*

A **GUI Layer** Component is attached to a **Camera** to enable **rendering** of 2D GUIs.

When a GUI Layer is attached to a Camera it will render all GUI Textures and GUI Texts in the **scene**. GUI Layers do not affect UnityGUI in any way.

You can enable and disable rendering GUI in a single camera by clicking on the check box of the GUI Layer in the **Inspector**.

Shader Reference

Leave feedback

Shaders in Unity can be written in one of three different ways:

as **surface shaders** ,
as **vertex and fragment shaders** or
as **fixed function shaders** .
The shader tutorial can guide you on choosing the right type for your needs.

Regardless of which type you choose, the actual meat of the shader code will always be wrapped in a language called **ShaderLab**, which is used to organize the shader structure. It looks like this:

```
Shader "MyShader" {
    Properties {
        _MyTexture ("My Texture", 2D) = "white" { }
        // other properties like colors or vectors go here as well
    }
    SubShader {
        // here goes the 'meat' of your
        // - surface shader or
        // - vertex and program shader or
        // - fixed function shader
    }
    SubShader {
        // here goes a simpler version of the SubShader
        // above than can run on older graphics cards
    }
}
```

We recommend that you start by reading about some basic concepts of the ShaderLab syntax in the sections listed below and then to move on to read about surface shaders or vertex and **fragment shaders** in other sections. Since fixed function shaders are written using ShaderLab only, you will find more information about them in the ShaderLab reference itself.

The reference below includes plenty of examples for the different types of shaders. For even more examples of surface shaders in particular, you can get the source of Unity's built-in shaders from the Resources section. Unity's post-processing effects allows you to create many interesting effects with shaders.

Read on for shader reference, and check out the shader tutorial as well!

See Also

Writing Surface Shaders.
Writing vertex and fragment shaders.
ShaderLab Syntax Reference.

Shader Assets.
Advanced ShaderLab Topics.

Writing Surface Shaders

Leave feedback

Writing **shaders** that interact with lighting is complex. There are different light types, different shadow options, different **rendering** paths (forward and deferred rendering), and the shader should somehow handle all that complexity.

Surface Shaders in Unity is a code generation approach that makes it much easier to write lit shaders than using low level vertex/pixel shader programs. Note that there are no custom languages, magic or ninjas involved in Surface Shaders; it just generates all the repetitive code that would have to be written by hand. You still write shader code in HLSL.

For some examples, take a look at Surface Shader Examples and Surface Shader Custom Lighting Examples.

How it works

You define a "surface function" that takes any UVs or data you need as input, and fills in output structure `SurfaceOutput`. SurfaceOutput basically describes *properties of the surface* (it's albedo color, normal, emission, specularity etc.). You write this code in HLSL.

Surface Shader compiler then figures out what inputs are needed, what outputs are filled and so on, and generates actual vertex&pixel shaders, as well as rendering passes to handle forward and deferred rendering.

Standard output structure of surface shaders is this:

```
struct SurfaceOutput
{
    fixed3 Albedo;  // diffuse color
    fixed3 Normal;  // tangent space normal, if written
    fixed3 Emission;
    half Specular;  // specular power in 0..1 range
    fixed Gloss;    // specular intensity
    fixed Alpha;    // alpha for transparencies
};
```

In Unity 5, surface shaders can also use physically based lighting models. Built-in Standard and StandardSpecular lighting models (see below) use these output structures respectively:

```
struct SurfaceOutputStandard
{
    fixed3 Albedo;      // base (diffuse or specular) color
    fixed3 Normal;      // tangent space normal, if written
    half3 Emission;
    half Metallic;      // 0=non-metal, 1=metal
    half Smoothness;    // 0=rough, 1=smooth
    half Occlusion;     // occlusion (default 1)
    fixed Alpha;        // alpha for transparencies
};
struct SurfaceOutputStandardSpecular
{
    fixed3 Albedo;      // diffuse color
    fixed3 Specular;    // specular color
```

```
    fixed3 Normal;      // tangent space normal, if written
    half3 Emission;
    half Smoothness;    // 0=rough, 1=smooth
    half Occlusion;     // occlusion (default 1)
    fixed Alpha;        // alpha for transparencies
};
```

Samples

See Surface Shader Examples, Surface Shader Custom Lighting Examples and Surface Shader Tessellation pages.

Surface Shader compile directives

Surface shader is placed inside `CGPROGRAM..ENDCG` block, just like any other shader. The differences are:

It must be placed inside SubShader block, not inside Pass. Surface shader will compile into multiple passes itself.
It uses `#pragma surface ...` directive to indicate it's a surface shader.
The `#pragma surface` directive is:

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

Required parameters

`surfaceFunction` - which Cg function has surface shader code. The function should have the form of `void surf (Input IN, inout SurfaceOutput o)`, where Input is a structure you have defined. Input should contain any texture coordinates and extra automatic variables needed by surface function.
`lightModel` - lighting model to use. Built-in ones are physically based `Standard` and `StandardSpecular`, as well as simple non-physically based `Lambert` (diffuse) and `BlinnPhong` (specular). See Custom Lighting Models page for how to write your own.
`Standard` lighting model uses `SurfaceOutputStandard` output struct, and matches the Standard (metallic workflow) shader in Unity.
`StandardSpecular` lighting model uses `SurfaceOutputStandardSpecular` output struct, and matches the Standard (specular setup) shader in Unity.
`Lambert` and `BlinnPhong` lighting models are not physically based (coming from Unity 4.x), but the shaders using them can be faster to render on low-end hardware.

Optional parameters

Transparency and alpha testing is controlled by `alpha` and `alphatest` directives. Transparency can typically be of two kinds: traditional alpha blending (used for fading objects out) or more physically plausible "premultiplied blending" (which allows semitransparent surfaces to retain proper specular reflections). Enabling semitransparency makes the generated surface shader code contain blending commands; whereas enabling alpha cutout will do a fragment discard in the generated **pixel** shader, based on the given variable.

`alpha` or `alpha:auto` - Will pick fade-transparency (same as `alpha:fade`) for simple lighting functions, and premultiplied transparency (same as `alpha:premul`) for physically based lighting functions.
`alpha:blend` - Enable alpha blending.

`alpha:fade` - Enable traditional fade-transparency.
`alpha:premul` - Enable premultiplied alpha transparency.
`alphatest:VariableName` - Enable alpha cutout transparency. Cutoff value is in a float variable with VariableName. You'll likely also want to use `addshadow` directive to generate proper shadow caster pass.
`keepalpha` - By default opaque surface shaders write 1.0 (white) into alpha channel, no matter what's output in the Alpha of output struct or what's returned by the lighting function. Using this option allows keeping lighting function's alpha value even for opaque surface shaders.
`decal:add` - Additive decal shader (e.g. terrain AddPass). This is meant for objects that lie atop of other surfaces, and use additive blending. See Surface Shader Examples
`decal:blend` - Semitransparent decal shader. This is meant for objects that lie atop of other surfaces, and use alpha blending. See Surface Shader Examples
Custom modifier functions can be used to alter or compute incoming vertex data, or to alter final computed fragment color.

`vertex:VertexFunction` - Custom vertex modification function. This function is invoked at start of generated **vertex shader**, and can modify or compute per-vertex data. See Surface Shader Examples.
`finalcolor:ColorFunction` - Custom final color modification function. See Surface Shader Examples.
`finalgbuffer:ColorFunction` - Custom deferred path for altering gbuffer content.
`finalprepass:ColorFunction` - Custom prepass base path.
Shadows and Tessellation - additional directives can be given to control how shadows and tessellation is handled.

`addshadow` - Generate a shadow caster pass. Commonly used with custom vertex modification, so that shadow casting also gets any procedural vertex animation. Often shaders don't need any special shadows handling, as they can just use shadow caster pass from their fallback.
`fullforwardshadows` - Support all light shadow types in Forward **rendering path**. By default shaders only support shadows from one directional light in **forward rendering** (to save on internal shader variant count). If you need point or spot light shadows in forward rendering, use this directive.
`tessellate:TessFunction` - use DX11 GPU tessellation; the function computes tessellation factors. See Surface Shader Tessellation for details.
Code generation options - by default generated surface shader code tries to handle all possible lighting/shadowing/**lightmap** scenarios. However in some cases you know you won't need some of them, and it is possible to adjust generated code to skip them. This can result in smaller shaders that are faster to load.

`exclude_path:deferred`, `exclude_path:forward`, `exclude_path:prepass` - Do not generate passes for given rendering path (Deferred Shading, Forward and Legacy Deferred respectively).
`noshadow` - Disables all shadow receiving support in this shader.
`noambient` - Do not apply any ambient lighting or **light probes**.
`novertexlights` - Do not apply any light probes or per-vertex lights in Forward rendering.
`nolightmap` - Disables all lightmapping support in this shader.
`nodynlightmap` - Disables runtime dynamic global illumination support in this shader.
`nodirlightmap` - Disables directional lightmaps support in this shader.
`nofog` - Disables all built-in Fog support.
`nometa` - Does not generate a "meta" pass (that's used by lightmapping & dynamic global illumination to extract surface information).
`noforwardadd` - Disables Forward rendering additive pass. This makes the shader support one full directional light, with all other lights computed per-vertex/SH. Makes shaders smaller as well.
`nolppv` - Disables **Light Probe Proxy Volume** support in this shader.
`noshadowmask` - Disables Shadowmask support for this shader (both Shadowmask and Distance Shadowmask).
Miscellaneous options

`softvegetation` - Makes the surface shader only be rendered when Soft Vegetation is on.
`interpolateview` - Compute view direction in the vertex shader and interpolate it; instead of computing it in the pixel shader. This can make the pixel shader faster, but uses up one more texture interpolator.
`halfasview` - Pass half-direction vector into the lighting function instead of view-direction. Half-direction will be computed and normalized per vertex. This is faster, but not entirely correct.
`approxview` - Removed in Unity 5.0. Use `interpolateview` instead.
`dualforward` - Use dual lightmaps in forward rendering path.
`dithercrossfade` - Makes the surface shader support dithering effects. You can then apply this shader to **GameObjects** that use an LOD Group **component** configured for cross-fade transition mode.
To see what exactly is different from using different options above, it can be helpful to use "Show Generated Code" button in the Shader Inspector.

Surface Shader input structure

The input structure `Input` generally has any texture coordinates needed by the shader. Texture coordinates must be named "`uv`" followed by texture name (or start it with "`uv2`" to use second texture coordinate set).

Additional values that can be put into Input structure:

`float3 viewDir` - contains view direction, for computing Parallax effects, rim lighting **etc**.
`float4` with `COLOR` semantic - contains interpolated per-vertex color.
`float4 screenPos` - contains screen space position for reflection or screenspace effects. Note that this is not suitable for GrabPass; you need to compute custom UV yourself using `ComputeGrabScreenPos` function.
`float3 worldPos` - contains world space position.
`float3 worldRefl` - contains world reflection vector *if surface shader does not write to o.Normal*. See Reflect-Diffuse shader for example.
`float3 worldNormal` - contains world normal vector *if surface shader does not write to o.Normal*.
`float3 worldRefl; INTERNAL_DATA` - contains world reflection vector *if surface shader writes to o.Normal*. To get the reflection vector based on per-pixel normal map, use `WorldReflectionVector (IN, o.Normal)`. See Reflect-Bumped shader for example.
`float3 worldNormal; INTERNAL_DATA` - contains world normal vector *if surface shader writes to o.Normal*. To get the normal vector based on per-pixel normal map, use `WorldNormalVector (IN, o.Normal)`.

Surface Shaders and DirectX 11 HLSL syntax

Currently some parts of surface shader compilation pipeline do not understand DirectX 11-specific HLSL syntax, so if you're using HLSL features like StructuredBuffers, RWTextures and other non-DX9 syntax, you have to wrap it into a DX11-only preprocessor macro.

See Platform Specific Differences and Shading Language pages for details.

2017–06–08 Page published with limited editorial review

`noshadowmask` added in 5.6

Surface Shader examples

Leave feedback

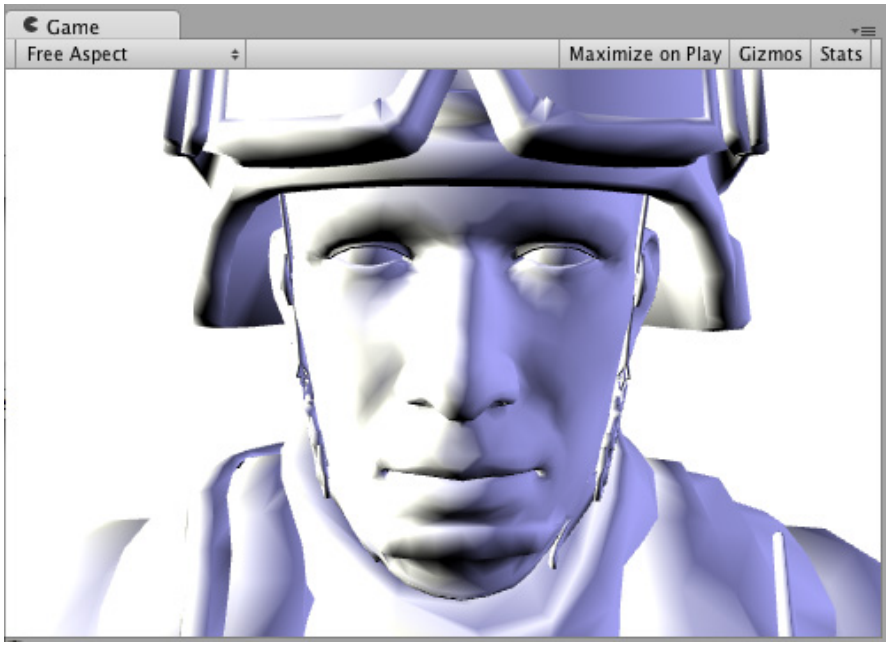
The Surface Shaders examples on this page show you how to use the built-in lighting models. For examples on how to implement custom lighting models, see Surface Shader Lighting Examples.

Simple shader example

We'll start with a very simple Shader and build up on that. Here's a Shader that sets the surface color to "white". It uses the built-in Lambert (diffuse) lighting model.

```
Shader "Example/Diffuse Simple" {
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float4 color : COLOR;
    };
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = 1;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Here's how it looks like on a model with two Lights set up:



Texture

An all-white object is quite boring, so let's add a Texture. We'll add a Properties block to the Shader, so we get a Texture selector in our Material. Other changes are in bold below.

```
Shader "Example/Diffuse Texture" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
    };
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Normal mapping

Let's add some normal mapping:

```
Shader "Example/Diffuse Bump" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _BumpMap ("Bumpmap", 2D) = "bump" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
```

```
      CGPROGRAM
      #pragma surface surf Lambert
      struct Input {
        float2 uv_MainTex;
        float2 uv_BumpMap;
      };
      sampler2D _MainTex;
      sampler2D _BumpMap;
      void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
      }
      ENDCG
    }
    Fallback "Diffuse"
  }
```

Rim Lighting

Now, try to add some Rim Lighting to highlight the edges of a **GameObject**. We'll add some emissive light based on the angle between surface normal and view direction. For that, we'll use the built-in `viewDir` Surface Shader variable.

```
  Shader "Example/Rim" {
    Properties {
      _MainTex ("Texture", 2D) = "white" {}
      _BumpMap ("Bumpmap", 2D) = "bump" {}
      _RimColor ("Rim Color", Color) = (0.26,0.19,0.16,0.0)
      _RimPower ("Rim Power", Range(0.5,8.0)) = 3.0
    }
    SubShader {
      Tags { "RenderType" = "Opaque" }
      CGPROGRAM
```

```
      #pragma surface surf Lambert
      struct Input {
          float2 uv_MainTex;
          float2 uv_BumpMap;
          float3 viewDir;
      };
      sampler2D _MainTex;
      sampler2D _BumpMap;
      float4 _RimColor;
      float _RimPower;
      void surf (Input IN, inout SurfaceOutput o) {
          o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
          o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
          half rim = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
          o.Emission = _RimColor.rgb * pow (rim, _RimPower);
      }
      ENDCG
    }
    Fallback "Diffuse"
  }
```

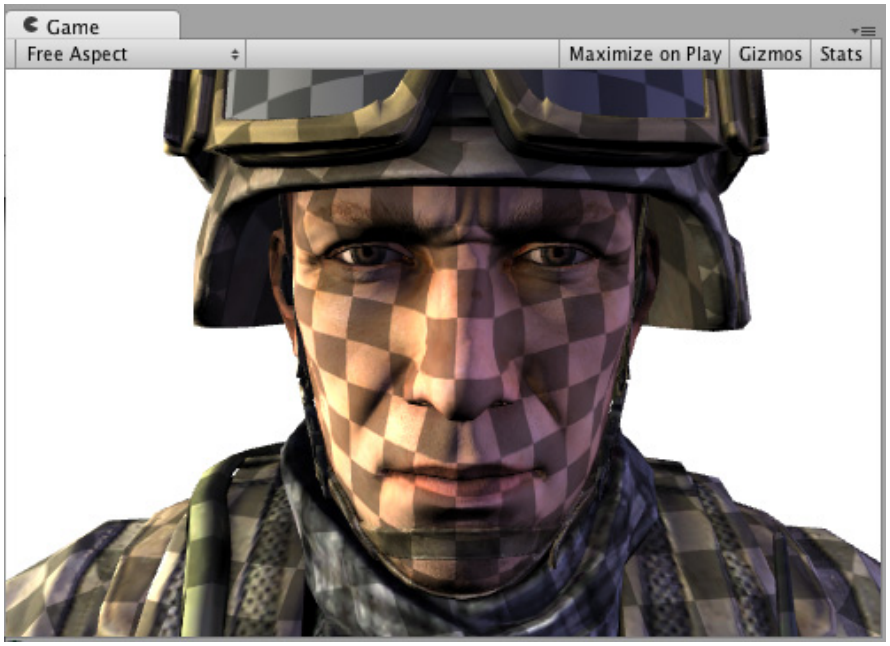
Detail Texture

For a different effect, let's add a Detail Texture that is combined with the base Texture. A Detail Texture usually uses the same UVs but different Tiling in the Material, so we need to use different input UV coordinates.

```
  Shader "Example/Detail" {
    Properties {
      _MainTex ("Texture", 2D) = "white" {}
      _BumpMap ("Bumpmap", 2D) = "bump" {}
      _Detail ("Detail", 2D) = "gray" {}
    }
```

```
    SubShader {
      Tags { "RenderType" = "Opaque" }
      CGPROGRAM
      #pragma surface surf Lambert
      struct Input {
          float2 uv_MainTex;
          float2 uv_BumpMap;
          float2 uv_Detail;
      };
      sampler2D _MainTex;
      sampler2D _BumpMap;
      sampler2D _Detail;
      void surf (Input IN, inout SurfaceOutput o) {
          o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
          o.Albedo *= tex2D (_Detail, IN.uv_Detail).rgb * 2;
          o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
      }
      ENDCG
    }
    Fallback "Diffuse"
  }
```

Using a Texture checker does not always make much practical sense, but in this example it is used to illustrate what happens:



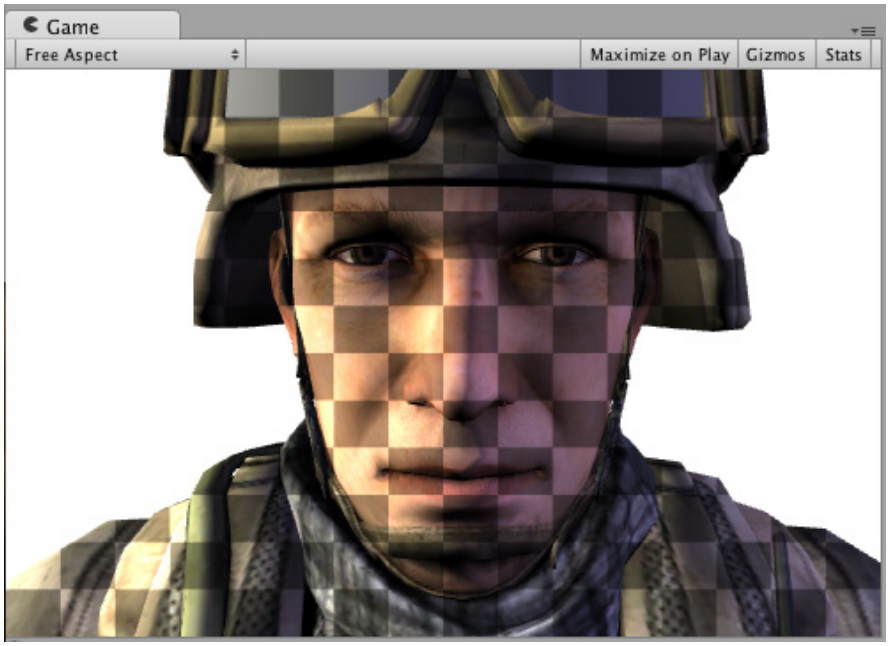
Detail Texture in Screen Space

A Detail Texture in screen space does not make practical sense for a soldier head model, but here it is used to illustrate how a built-in `screenPos` input might be used:

```
  Shader "Example/ScreenPos" {
    Properties {
      _MainTex ("Texture", 2D) = "white" {}
```

```
      _Detail ("Detail", 2D) = "gray" {}
    }
    SubShader {
      Tags { "RenderType" = "Opaque" }
      CGPROGRAM
      #pragma surface surf Lambert
      struct Input {
          float2 uv_MainTex;
          float4 screenPos;
      };
      sampler2D _MainTex;
      sampler2D _Detail;
      void surf (Input IN, inout SurfaceOutput o) {
          o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
          float2 screenUV = IN.screenPos.xy / IN.screenPos.w;
          screenUV *= float2(8,6);
          o.Albedo *= tex2D (_Detail, screenUV).rgb * 2;
      }
      ENDCG
    }
    Fallback "Diffuse"
  }
```

The normal mapping has been removed from the Shader above, just to make it shorter:



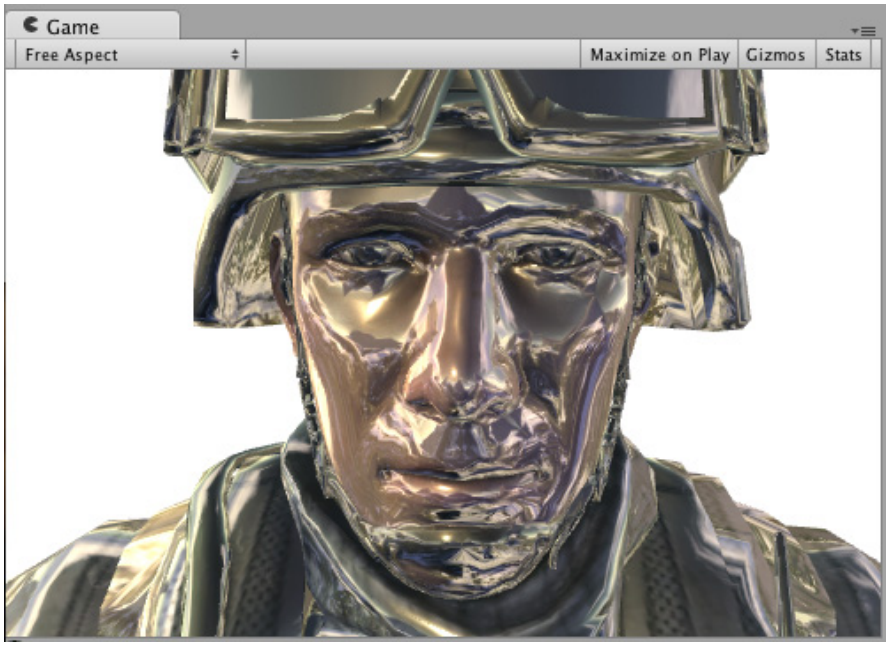
Cubemap Reflection

Here's a Shader that does cubemapped reflection using built-in `worldRefl` input. It's very similar to built-in Reflective/**Diffuse Shader**:

```
  Shader "Example/WorldRefl" {
    Properties {
```

```
    _MainTex ("Texture", 2D) = "white" {}
    _Cube ("Cubemap", CUBE) = "" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
        float3 worldRefl;
    };
    sampler2D _MainTex;
    samplerCUBE _Cube;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb * 0.5;
        o.Emission = texCUBE (_Cube, IN.worldRefl).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Because it assigns the reflection color as **Emission**, we get a very shiny soldier:

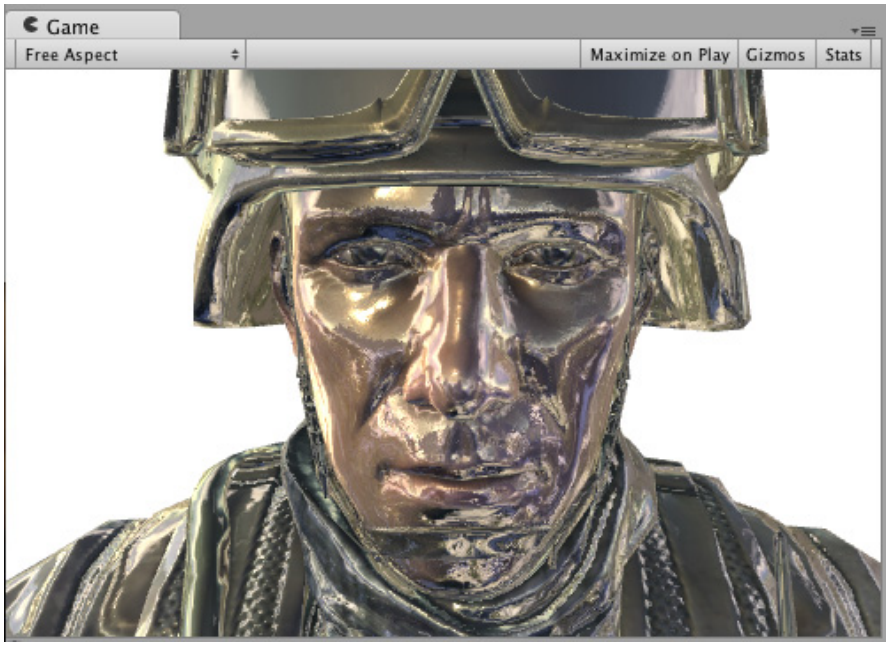


If you want to do reflections that are affected by normal maps, it needs to be slightly more involved: `INTERNAL_DATA` needs to be added to the `Input` structure, and `WorldReflectionVector` function used to compute per-pixel reflection vector after you've written the Normal output.

```
Shader "Example/WorldRefl Normalmap" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _BumpMap ("Bumpmap", 2D) = "bump" {}
```

```
        _Cube ("Cubemap", CUBE) = "" {}
    }
    SubShader {
      Tags { "RenderType" = "Opaque" }
      CGPROGRAM
      #pragma surface surf Lambert
      struct Input {
          float2 uv_MainTex;
          float2 uv_BumpMap;
          float3 worldRefl;
          INTERNAL_DATA
      };
      sampler2D _MainTex;
      sampler2D _BumpMap;
      samplerCUBE _Cube;
      void surf (Input IN, inout SurfaceOutput o) {
          o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb * 0.5;
          o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
          o.Emission = texCUBE (_Cube, WorldReflectionVector (IN, o.Normal)).rgb;
      }
      ENDCG
    }
    Fallback "Diffuse"
  }
```

Here's a normal-mapped shiny soldier:



Slices via World Space Position

Here's a Shader that "slices" the GameObject by discarding **pixels** in nearly horizontal rings. It does this by using the `clip()` Cg/HLSL function based on the world position of a pixel. We'll use the built-in `worldPos` Surface Shader variable.

```
Shader "Example/Slices" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _BumpMap ("Bumpmap", 2D) = "bump" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    Cull Off
    CGPROGRAM
    #pragma surface surf Lambert
    struct Input {
        float2 uv_MainTex;
        float2 uv_BumpMap;
        float3 worldPos;
    };
    sampler2D _MainTex;
    sampler2D _BumpMap;
    void surf (Input IN, inout SurfaceOutput o) {
        clip (frac((IN.worldPos.y+IN.worldPos.z*0.1) * 5) - 0.5);
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Normal Extrusion with Vertex Modifier

It is possible to use a “vertex modifier” function that will modify the incoming vertex data in the **vertex Shader**. This can be used for things like procedural animation and extrusion along normals. Surface Shader compilation directive `vertex:functionName`

is used for that, with a function that takes `inout appdata_full` parameter.

Here's a Shader that moves vertices along their normals by the amount specified in the Material:

```
Shader "Example/Normal Extrusion" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _Amount ("Extrusion Amount", Range(-1,1)) = 0.5
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert vertex:vert
    struct Input {
        float2 uv_MainTex;
    };
    float _Amount;
    void vert (inout appdata_full v) {
        v.vertex.xyz += v.normal * _Amount;
    }
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Moving vertices along their normals makes a fat soldier:

Custom data computed per-vertex

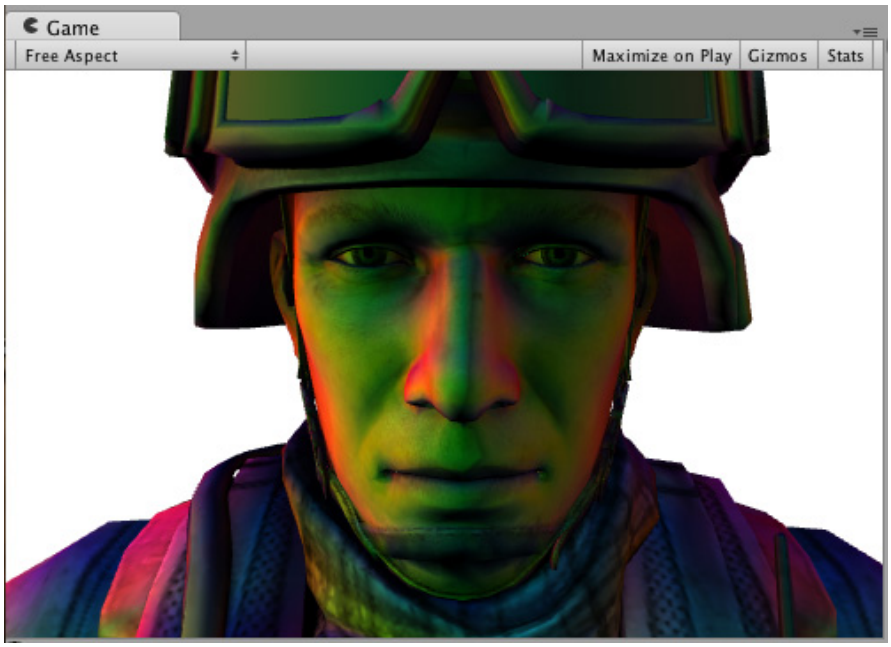
Using a vertex modifier function, it is also possible to compute custom data in a vertex Shader, which then will be passed to the Surface Shader function per-pixel. The same compilation directive `vertex:functionName` is used, but the function should take two parameters: `inout appdata_full` and `out Input`. You can fill in any Input member that is not a built-in value there.

Note: Custom Input members used in this way must not have names beginning with 'uv' or they won't work properly.

The example below defines a custom `float3 customColor` member, which is computed in a vertex function:

```
Shader "Example/Custom Vertex Data" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert vertex:vert
    struct Input {
        float2 uv_MainTex;
        float3 customColor;
    };
    void vert (inout appdata_full v, out Input o) {
        UNITY_INITIALIZE_OUTPUT(Input,o);
        o.customColor = abs(v.normal);
    }
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        o.Albedo *= IN.customColor;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

In this example `customColor` is set to the absolute value of the normal:



More practical uses could be computing any per-vertex data that is not provided by built-in Input variables; or optimizing Shader computations. For example, it's possible to compute Rim lighting at the GameObject's vertices, instead of doing that in the Surface Shader per-pixel.

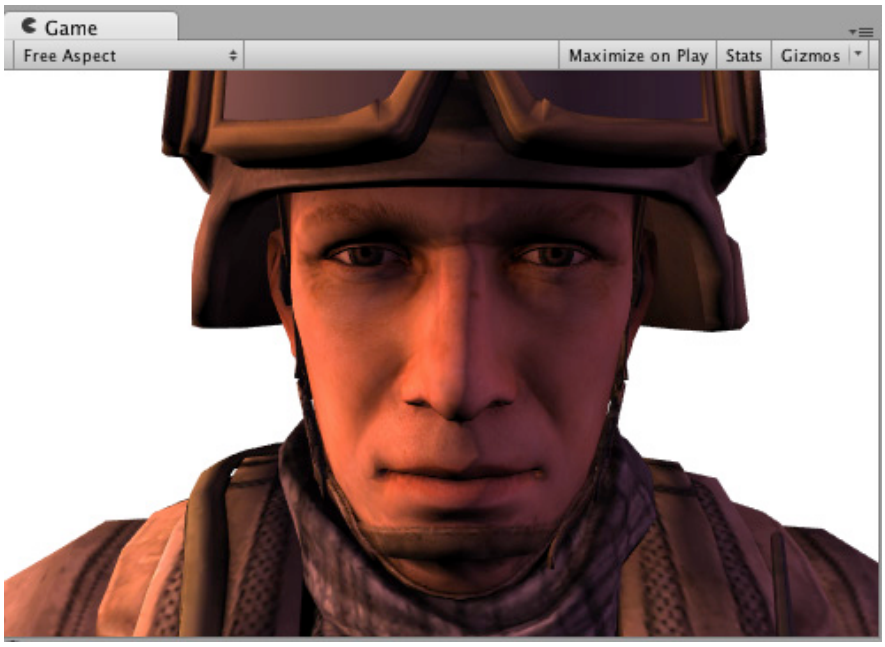
Final Color Modifier

It is possible to use a "final color modifier" function that will modify the final color computed by the Shader.The Surface Shader compilation directive `finalcolor:functionName` is used for this, with a function that takes `Input IN, SurfaceOutput o, inout fixed4 color` parameters.

Here's a simple Shader that applies tint to the final color. This is different from just applying tint to the surface Albedo color: this tint will also affect any color that comes from **Lightmaps**, **Light Probes** and similar extra sources.

```
Shader "Example/Tint Final Color" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _ColorTint ("Tint", Color) = (1.0, 0.6, 0.6, 1.0)
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert finalcolor:mycolor
    struct Input {
        float2 uv_MainTex;
    };
    fixed4 _ColorTint;
    void mycolor (Input IN, SurfaceOutput o, inout fixed4 color)
    {
        color *= _ColorTint;
    }
    sampler2D _MainTex;
    void surf (Input IN, inout SurfaceOutput o) {
         o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
```

```
    }
    Fallback "Diffuse"
  }
```



Custom Fog with Final Color Modifier

A common case for using final color modifier (see above) would be implementing completely custom Fog in **forward rendering**. Fog needs to affect the final computed pixel Shader color, which is exactly what the `finalcolor` modifier does.

Here's a Shader that applies fog tint based on the distance from screen center. This combines the vertex modifier with the custom vertex data (`fog`) and the final color modifier. When used in the forward **rendering** additive pass, the Fog needs to fade to black. This example handles that and performs a check for `UNITY_PASS_FORWARDADD`.

```
Shader "Example/Fog via Final Color" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
    _FogColor ("Fog Color", Color) = (0.3, 0.4, 0.7, 1.0)
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert finalcolor:mycolor vertex:myvert
    struct Input {
        float2 uv_MainTex;
        half fog;
    };
    void myvert (inout appdata_full v, out Input data)
    {
        UNITY_INITIALIZE_OUTPUT(Input,data);
        float4 hpos = UnityObjectToClipPos(v.vertex);
        hpos.xy/=hpos.w;
        data.fog = min (1, dot (hpos.xy, hpos.xy)*0.5);
```

```
      }
      fixed4 _FogColor;
      void mycolor (Input IN, SurfaceOutput o, inout fixed4 color)
      {
          fixed3 fogColor = _FogColor.rgb;
          #ifdef UNITY_PASS_FORWARDADD
          fogColor = 0;
          #endif
          color.rgb = lerp (color.rgb, fogColor, IN.fog);
      }
      sampler2D _MainTex;
      void surf (Input IN, inout SurfaceOutput o) {
           o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
      }
      ENDCG
    }
    Fallback "Diffuse"
  }
```

Linear Fog

```
Shader "Example/Linear Fog" {
  Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType"="Opaque" }
    LOD 200

    CGPROGRAM
    #pragma surface surf Lambert finalcolor:mycolor vertex:myvert
    #pragma multi_compile_fog
```

```
    sampler2D _MainTex;
    uniform half4 unity_FogStart;
    uniform half4 unity_FogEnd;

    struct Input {
      float2 uv_MainTex;
      half fog;
    };

    void myvert (inout appdata_full v, out Input data) {
      UNITY_INITIALIZE_OUTPUT(Input,data);
      float pos = length(UnityObjectToViewPos(v.vertex).xyz);
      float diff = unity_FogEnd.x - unity_FogStart.x;
      float invDiff = 1.0f / diff;
      data.fog = clamp ((unity_FogEnd.x - pos) * invDiff, 0.0, 1.0);
    }
    void mycolor (Input IN, SurfaceOutput o, inout fixed4 color) {
      #ifdef UNITY_PASS_FORWARDADD
        UNITY_APPLY_FOG_COLOR(IN.fog, color, float4(0,0,0,0));
      #else
        UNITY_APPLY_FOG_COLOR(IN.fog, color, unity_FogColor);
      #endif
    }

    void surf (Input IN, inout SurfaceOutput o) {
      half4 c = tex2D (_MainTex, IN.uv_MainTex);
      o.Albedo = c.rgb;
      o.Alpha = c.a;
    }
    ENDCG
  }
  FallBack "Diffuse"
}
```

Decals

Decals are commonly used to add details to Materials at run time (for example, bullet impacts). They are especially useful in deferred rendering, because they alter the GBuffer before it is lit, therefore saving on performance.

In a typical scenario, Decals should be rendered after the opaque objects and should not be shadow casters, as seen in the **ShaderLab** "Tags" in the example below.

```
Shader "Example/Decal" {
  Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType"="Opaque" "Queue"="Geometry+1" "ForceNoShadowCasting"="True" }
    LOD 200
```

```
    Offset -1, -1

    CGPROGRAM
    #pragma surface surf Lambert decal:blend

    sampler2D _MainTex;

    struct Input {
      float2 uv_MainTex;
    };

    void surf (Input IN, inout SurfaceOutput o) {
        half4 c = tex2D (_MainTex, IN.uv_MainTex);
        o.Albedo = c.rgb;
        o.Alpha = c.a;
      }
    ENDCG
    }
}
```

Custom lighting models in Surface Shaders

Leave feedback

When writing Surface Shaders, you describe the properties of a surface (such as albedo color and normal), and a **Lighting Model** computes the lighting interaction.

There are two built-in lighting models: `Lambert` for diffuse lighting, and `BlinnPhong` for specular lighting. The *Lighting.cginc* file inside Unity defines these models (Windows: *<unity install path>/Data/CGIncludes/Lighting.cginc*; macOS: */Applications/Unity/Unity.app/Contents/CGIncludes/Lighting.cginc*).

Sometimes you might want to use a custom lighting model. You can do this with Surface **Shaders**. A lighting model is simply a couple of Cg/HLSL functions that match some conventions.

Declaring lighting models

A lighting model consists of regular functions with names that begin `Lighting`. You can declare them anywhere in your shader file, or one of the included files. The functions are:

`half4 Lighting<Name> (SurfaceOutput s, UnityGI gi);` Use this in **forward rendering** paths for light models that are *not* dependent on the view direction.

`half4 Lighting<Name> (SurfaceOutput s, half3 viewDir, UnityGI gi);` Use this in forward **rendering** paths for light models that *are* dependent on the view direction.

`half4 Lighting<Name>_Deferred (SurfaceOutput s, UnityGI gi, out half4 outDiffuseOcclusion, out half4 outSpecSmoothness, out half4 outNormal);` Use this in deferred lighting paths.

`half4 Lighting<Name>_PrePass (SurfaceOutput s, half4 light);` Use this in light prepass (legacy deferred) lighting paths.

Note that you don't need to declare all functions. A lighting model either uses view direction or it does not. Similarly, if the lighting model only works in forward rendering, do not declare the `_Deferred` or `_Prepass` function. This ensures that Shaders that use it only compile to forward rendering.

Custom GI

Declare the following function to customize the decoding **lightmap** data and probes:

`half4 Lighting<Name>_GI (SurfaceOutput s, UnityGIInput data, inout UnityGI gi);` Note that to decode standard Unity lightmaps and SH probes, you can use the built-in `DecodeLightmap` and `ShadeSHPerPixel` functions, as seen in `UnityGI_Base` in the *UnityGlobalIllumination.cginc* file inside Unity (Windows: *<unity install path>/Data/CGIncludes/UnityGlobalIllumination.cginc*; macOS: */Applications/Unity/Unity.app/Contents/CGIncludes/UnityGlobalIllumination.cginc_*).

Examples

See documentation on Surface Shader Lighting Examples for more information.

Surface Shader lighting examples

Leave feedback

This page provides examples of custom Surface Shader lighting models in Surface Shaders. For more general Surface **Shader** guidance, see Surface Shader Examples.

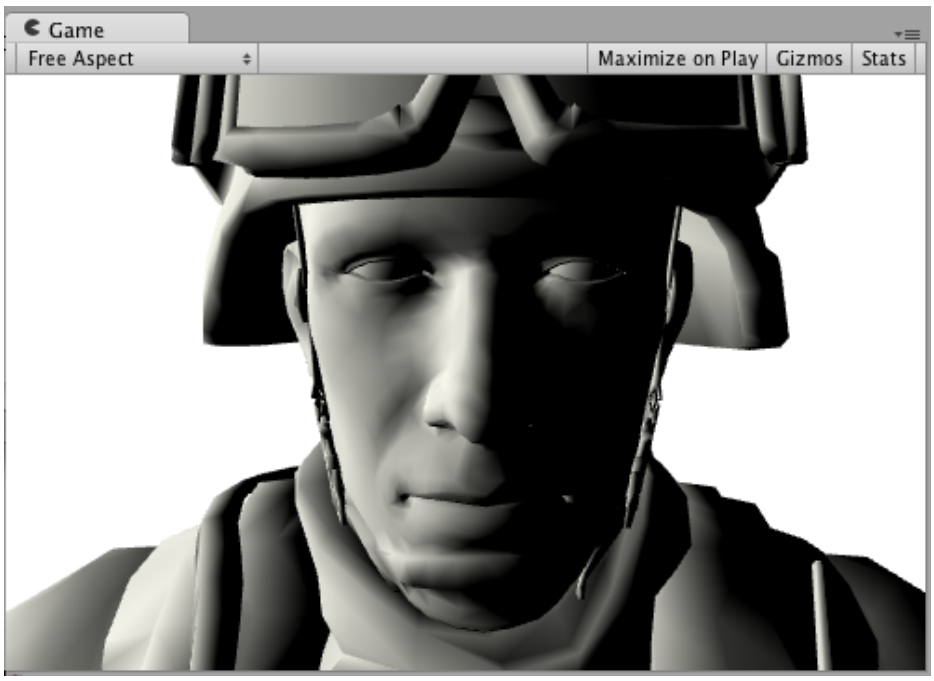
Because **Deferred Lighting** does not play well with some custom per-material lighting models, most of the examples below make the shaders compile to **Forward Rendering** only.

Diffuse

The following is an example of a shader that uses the built-in Lambert lighting model:

```
Shader "Example/Diffuse Texture" {
  Properties {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert

    struct Input {
        float2 uv_MainTex;
    };

    sampler2D _MainTex;

    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Here's how it looks like with a Texture and without a Texture, with one directional Light in the **Scene**:



The following example shows how to achieve the same result by writing a custom lighting model instead of using the built-in Lambert model.

To do this, you need to use a number of Surface Shader lighting model functions. Here's a simple Lambert one. Note that only the `CGPROGRAM` section changes; the surrounding Shader code is exactly the same:

```
Shader "Example/Diffuse Texture" {
    Properties {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
      #pragma surface surf SimpleLambert
```

```
      half4 LightingSimpleLambert (SurfaceOutput s, half3 lightDir, half atten) {
          half NdotL = dot (s.Normal, lightDir);
          half4 c;
          c.rgb = s.Albedo * _LightColor0.rgb * (NdotL * atten);
          c.a = s.Alpha;
          return c;
      }

    struct Input {
        float2 uv_MainTex;
    };

    sampler2D _MainTex;

    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
    }
    Fallback "Diffuse"
}
```

This simple Diffuse lighting model uses the `LightingSimpleLambert` function. It computes lighting by calculating a dot product between surface normal and light direction, and then applying light attenuation and color.

Diffuse Wrap

The following example shows **Wrapped Diffuse**, a modification of **Diffuse** lighting where illumination “wraps around” the edges of objects. It’s useful for simulating subsurface scattering effects. Only the `CGPROGRAM` section changes, so once again, the surrounding Shader code is omitted:

```
...ShaderLab code...
CGPROGRAM
#pragma surface surf WrapLambert

half4 LightingWrapLambert (SurfaceOutput s, half3 lightDir, half atten) {
    half NdotL = dot (s.Normal, lightDir);
    half diff = NdotL * 0.5 + 0.5;
    half4 c;
    c.rgb = s.Albedo * _LightColor0.rgb * (diff * atten);
    c.a = s.Alpha;
    return c;
}

struct Input {
    float2 uv_MainTex;
```

```
    };

    sampler2D _MainTex;
        void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
    ...ShaderLab code...
```

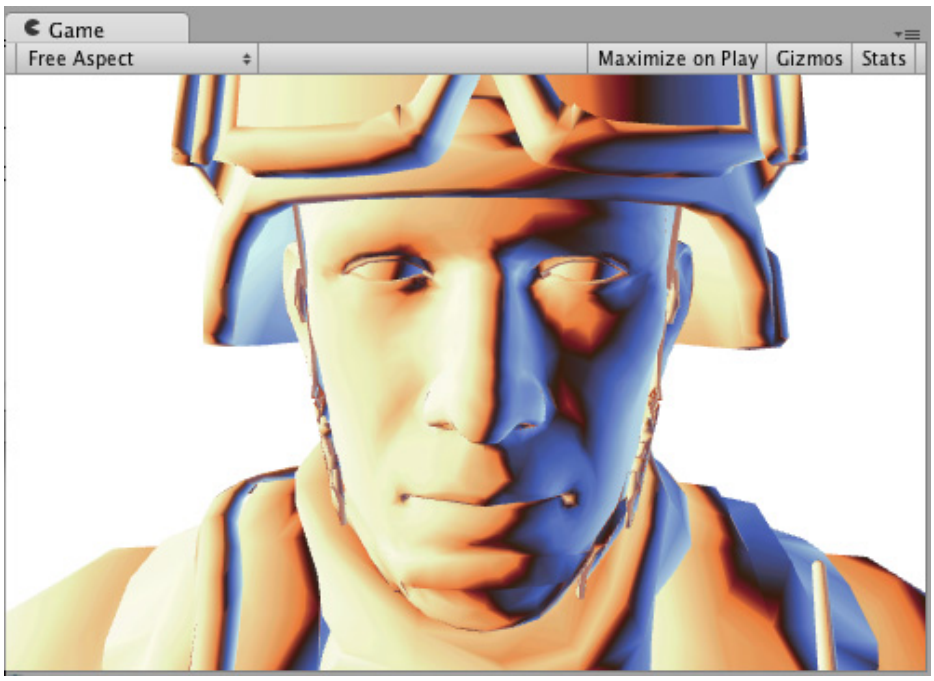
Here's how it looks like with a Texture and without a Texture, with one directional Light in the Scene:

Toon Ramp

The following example shows a “Ramp” lighting model that uses a Texture ramp to define how surfaces respond to the angles between the light and the normal. This can be used for a variety of effects, and is especially effective when used with **Toon** lighting.

```
...ShaderLab code...
CGPROGRAM
#pragma surface surf Ramp

sampler2D _Ramp;

half4 LightingRamp (SurfaceOutput s, half3 lightDir, half atten) {
    half NdotL = dot (s.Normal, lightDir);
    half diff = NdotL * 0.5 + 0.5;
    half3 ramp = tex2D (_Ramp, float2(diff)).rgb;
    half4 c;
    c.rgb = s.Albedo * _LightColor0.rgb * ramp * atten;
    c.a = s.Alpha;
    return c;
}

struct Input {
    float2 uv_MainTex;
};

sampler2D _MainTex;

void surf (Input IN, inout SurfaceOutput o) {
    o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
}
ENDCG
...ShaderLab code...
```

Here’s how it looks like with a Texture and without a Texture, with one directional Light in the Scene:



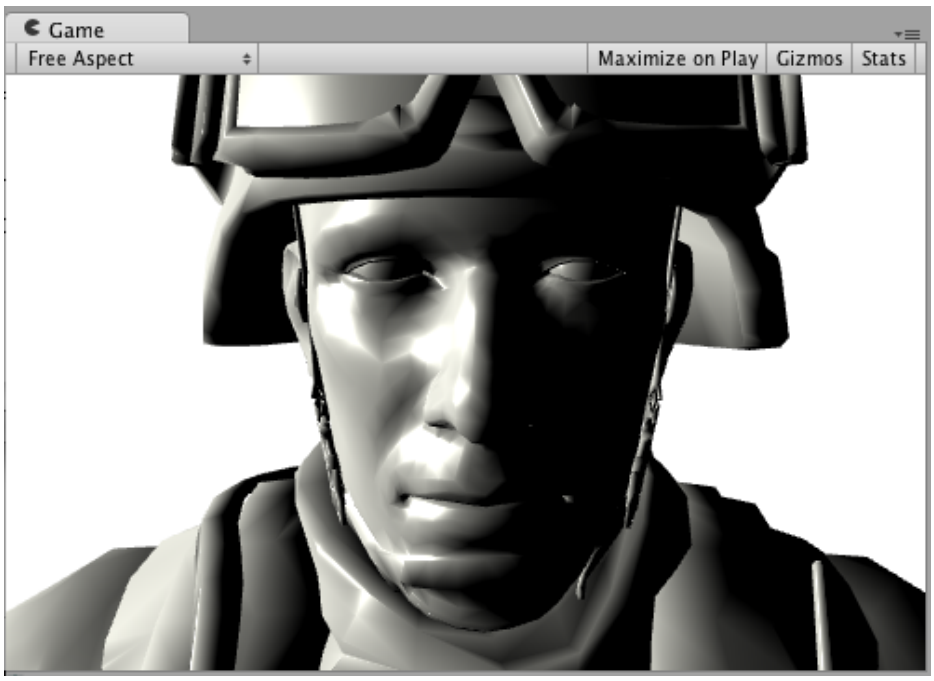
Simple Specular

The following example shows a simple specular lighting model, similar to the built-in BlinnPhong lighting model.

```
...ShaderLab code...
CGPROGRAM
#pragma surface surf SimpleSpecular

half4 LightingSimpleSpecular (SurfaceOutput s, half3 lightDir, half3 viewDir, half
    half3 h = normalize (lightDir + viewDir);

    half diff = max (0, dot (s.Normal, lightDir));
```

```
        float nh = max (0, dot (s.Normal, h));
        float spec = pow (nh, 48.0);

        half4 c;
        c.rgb = (s.Albedo * _LightColor0.rgb * diff + _LightColor0.rgb * spec) * atten,
        c.a = s.Alpha;
        return c;
    }

    struct Input {
        float2 uv_MainTex;
    };

    sampler2D _MainTex;

    void surf (Input IN, inout SurfaceOutput o) {
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    ENDCG
    ...ShaderLab code...
```

Here’s how it looks like with a Texture and without a Texture, with one directional Light in the Scene:



Custom GI

We'll start with a Shader that mimics Unity's built-in GI:

```
Shader "Example/CustomGI_ToneMapped" {
    Properties {
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
    }
    SubShader {
        Tags { "RenderType"="Opaque" }

        CGPROGRAM
        #pragma surface surf StandardDefaultGI
```

```
        #include "UnityPBSLighting.cginc"

        sampler2D _MainTex;

        inline half4 LightingStandardDefaultGI(SurfaceOutputStandard s, half3 viewI
        {
            return LightingStandard(s, viewDir, gi);
        }

        inline void LightingStandardDefaultGI_GI(
            SurfaceOutputStandard s,
            UnityGIInput data,
            inout UnityGI gi)
        {
            LightingStandard_GI(s, data, gi);
        }

        struct Input {
            float2 uv_MainTex;
        };

        void surf (Input IN, inout SurfaceOutputStandard o) {
            o.Albedo = tex2D(_MainTex, IN.uv_MainTex);
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Now, let’s add some tone mapping on top of the GI:

```
Shader "Example/CustomGI_ToneMapped" {
    Properties {
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Gain("Lightmap tone-mapping Gain", Float) = 1
        _Knee("Lightmap tone-mapping Knee", Float) = 0.5
        _Compress("Lightmap tone-mapping Compress", Float) = 0.33
    }
    SubShader {
        Tags { "RenderType"="Opaque" }

        CGPROGRAM
        #pragma surface surf StandardToneMappedGI

        #include "UnityPBSLighting.cginc"

        half _Gain;
```

```
        half _Knee;
        half _Compress;
        sampler2D _MainTex;

        inline half3 TonemapLight(half3 i) {
            i *= _Gain;
            return (i > _Knee) ? (((i - _Knee)*_Compress) + _Knee) : i;
        }

        inline half4 LightingStandardToneMappedGI(SurfaceOutputStandard s, half3 v
        {
            return LightingStandard(s, viewDir, gi);
        }

        inline void LightingStandardToneMappedGI_GI(
            SurfaceOutputStandard s,
            UnityGIInput data,
            inout UnityGI gi)
        {
            LightingStandard_GI(s, data, gi);

            gi.light.color = TonemapLight(gi.light.color);
            #ifdef DIRLIGHTMAP_SEPARATE
                #ifdef LIGHTMAP_ON
                    gi.light2.color = TonemapLight(gi.light2.color);
                #endif
                #ifdef DYNAMICLIGHTMAP_ON
                    gi.light3.color = TonemapLight(gi.light3.color);
                #endif
            #endif
            gi.indirect.diffuse = TonemapLight(gi.indirect.diffuse);
            gi.indirect.specular = TonemapLight(gi.indirect.specular);
        }

        struct Input {
            float2 uv_MainTex;
        };

        void surf (Input IN, inout SurfaceOutputStandard o) {
            o.Albedo = tex2D(_MainTex, IN.uv_MainTex);
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Surface Shaders with DX11 / OpenGL Core Tessellation

Leave feedback

Surface Shaders have some support for DirectX 11 / **OpenGL Core** GPU Tessellation. Idea is:

Tessellation is indicated by `tessellate:FunctionName` modifier. That function computes triangle edge and inside tessellation factors.
When tessellation is used, "vertex modifier" (`vertex:FunctionName`) is invoked *after* tessellation, for each generated vertex in the domain **shader**. Here you'd typically to displacement mapping.
Surface shaders can optionally compute phong tessellation to smooth model surface even without any displacement mapping.
Current limitations of tessellation support:

Only triangle domain - no **quads**, no isoline tessellation.
When you use tessellation, the shader is automatically compiled into the Shader Model 4.6 target, which prevents support for running on older graphics targets.

No GPU tessellation, displacement in the vertex modifier

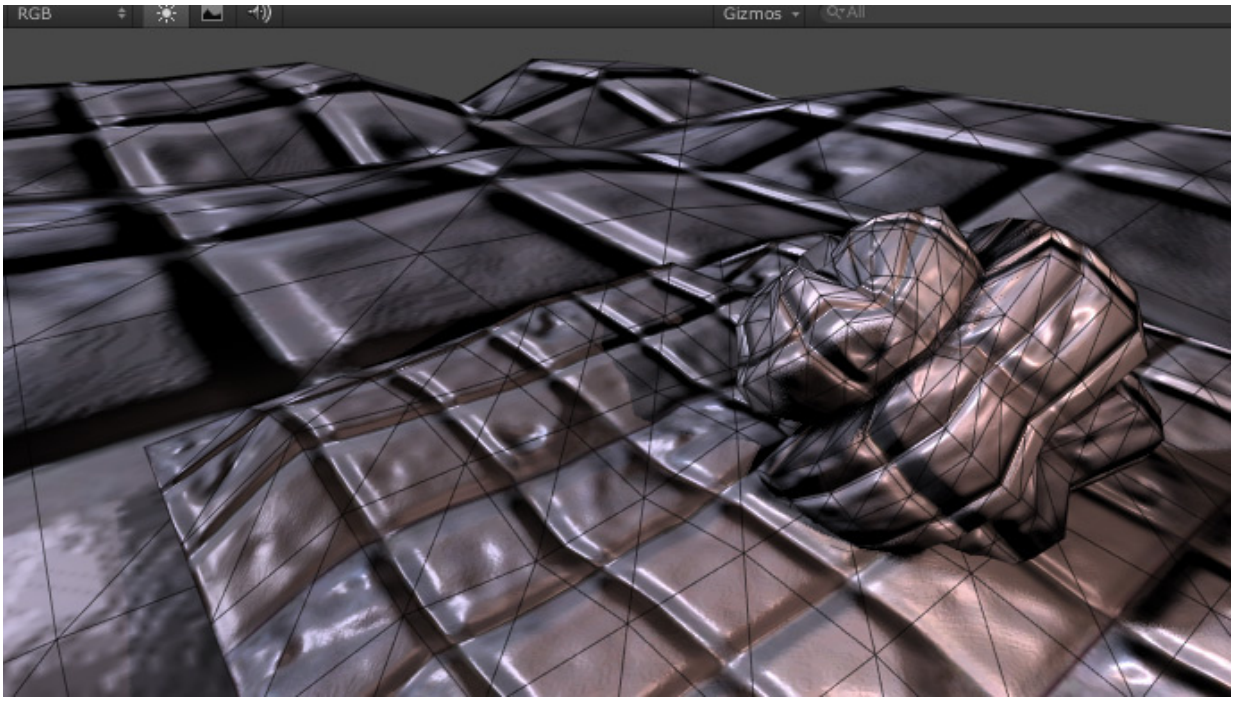
This next example shows a surface shader that does some displacement mapping *without* using tessellation. It just moves vertices along their normals based on the amount coming from a displacement map:

```
Shader "Tessellation Sample" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DispTex ("Disp Texture", 2D) = "gray" {}
        _NormalMap ("Normalmap", 2D) = "bump" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _Color ("Color", color) = (1,1,1,0)
        _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp nolig
        #pragma target 4.6

        struct appdata {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };

        sampler2D _DispTex;
        float _Displacement;

        void disp (inout appdata v)
        {
```

```
            float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
            v.vertex.xyz += v.normal * d;
        }

        struct Input {
            float2 uv_MainTex;
        };

        sampler2D _MainTex;
        sampler2D _NormalMap;
        fixed4 _Color;

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Specular = 0.2;
            o.Gloss = 1.0;
            o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

The above shader is fairly standard:

Vertex modifier `disp` samples the displacement map and moves vertices along their normals.
It uses custom “vertex data input” structure (`appdata`) instead of default `appdata_full`. This is not needed yet, but it’s more efficient for tessellation to use as small structure as possible.
Since our vertex data does not have 2nd UV coordinate, we add `nolightmap` directive to exclude **lightmaps**.
The image below displays some simple **GameObjects** with this shader applied.



Fixed amount of tessellation

If your model's faces are roughly the same size on screen, add a fixed amount of tesselation to the **Mesh** (the same tessellation level over the whole Mesh).

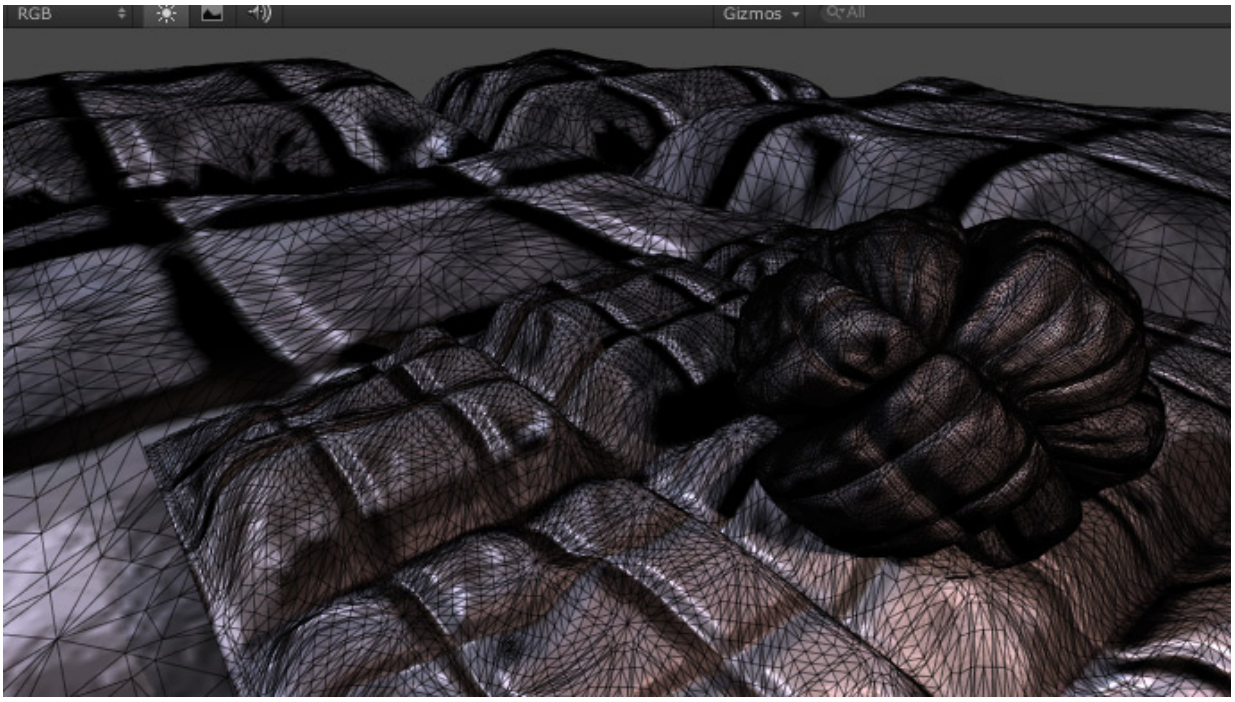
The following example script applies a fixed amount of tessellation.

```
Shader "Tessellation Sample" {
    Properties {
        _Tess ("Tessellation", Range(1,32)) = 4
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DispTex ("Disp Texture", 2D) = "gray" {}
        _NormalMap ("Normalmap", 2D) = "bump" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _Color ("Color", color) = (1,1,1,0)
        _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp tesse
        #pragma target 4.6

        struct appdata {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };
```

```
            float _Tess;

            float4 tessFixed()
            {
                return _Tess;
            }

            sampler2D _DispTex;
            float _Displacement;

            void disp (inout appdata v)
            {
                float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
                v.vertex.xyz += v.normal * d;
            }

            struct Input {
                float2 uv_MainTex;
            };

            sampler2D _MainTex;
            sampler2D _NormalMap;
            fixed4 _Color;

            void surf (Input IN, inout SurfaceOutput o) {
                half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
                o.Albedo = c.rgb;
                o.Specular = 0.2;
                o.Gloss = 1.0;
                o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
            }
            ENDCG
        }
        FallBack "Diffuse"
    }
```

In the example above, the `tessFixed` tessellation function returns four tessellation factors as a single float4 value: three factors for each edge of the triangle, and one factor for the inside of the triangle.

The example returns a constant value that is set in the Material properties.



Distance-based tessellation

You can also change tessellation level based on distance from the **camera**. For example, you could define two distance values:

The distance when tessellation is at maximum (for example, 10 meters).
The distance when the tessellation level gradually decreases (for example, 20 meters).

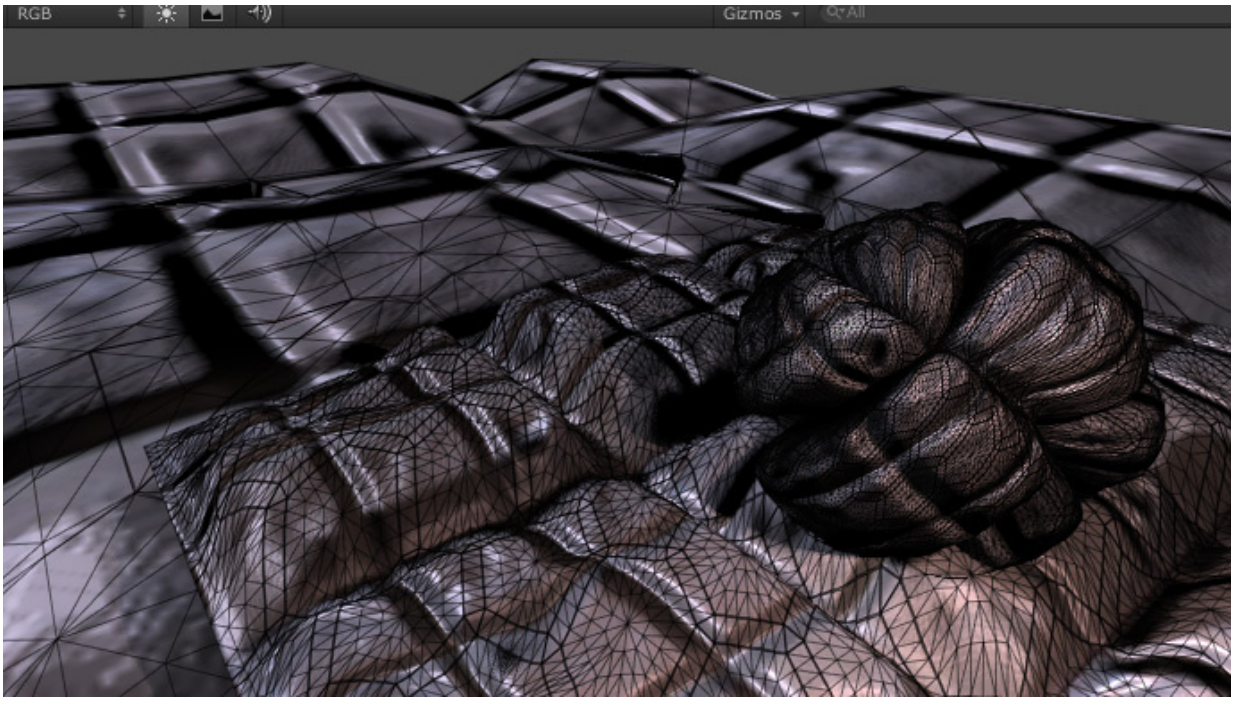
```
Shader "Tessellation Sample" {
    Properties {
        _Tess ("Tessellation", Range(1,32)) = 4
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DispTex ("Disp Texture", 2D) = "gray" {}
        _NormalMap ("Normalmap", 2D) = "bump" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _Color ("Color", color) = (1,1,1,0)
        _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp tesse
        #pragma target 4.6
        #include "Tessellation.cginc"

        struct appdata {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };
```

```
            float _Tess;

            float4 tessDistance (appdata v0, appdata v1, appdata v2) {
                float minDist = 10.0;
                float maxDist = 25.0;
                return UnityDistanceBasedTess(v0.vertex, v1.vertex, v2.vertex, minDist, ma
            }

            sampler2D _DispTex;
            float _Displacement;

            void disp (inout appdata v)
            {
                float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
                v.vertex.xyz += v.normal * d;
            }

            struct Input {
                float2 uv_MainTex;
            };

            sampler2D _MainTex;
            sampler2D _NormalMap;
            fixed4 _Color;

            void surf (Input IN, inout SurfaceOutput o) {
                half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
                o.Albedo = c.rgb;
                o.Specular = 0.2;
                o.Gloss = 1.0;
                o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
            }
            ENDCG
        }
        FallBack "Diffuse"
    }
```

Here, the tessellation function takes the vertex data of the three triangle corners before tessellation as its three parameters.

Unity needs this to compute tessellation levels, which depend on vertex positions.

The example includes a built-in helper file, Tessellation.cginc, and calls the `UnityDistanceBasedTess` function from the file to do all the work. This function computes the distance of each vertex to the camera and derives the final tessellation factors.



Edge length based tessellation

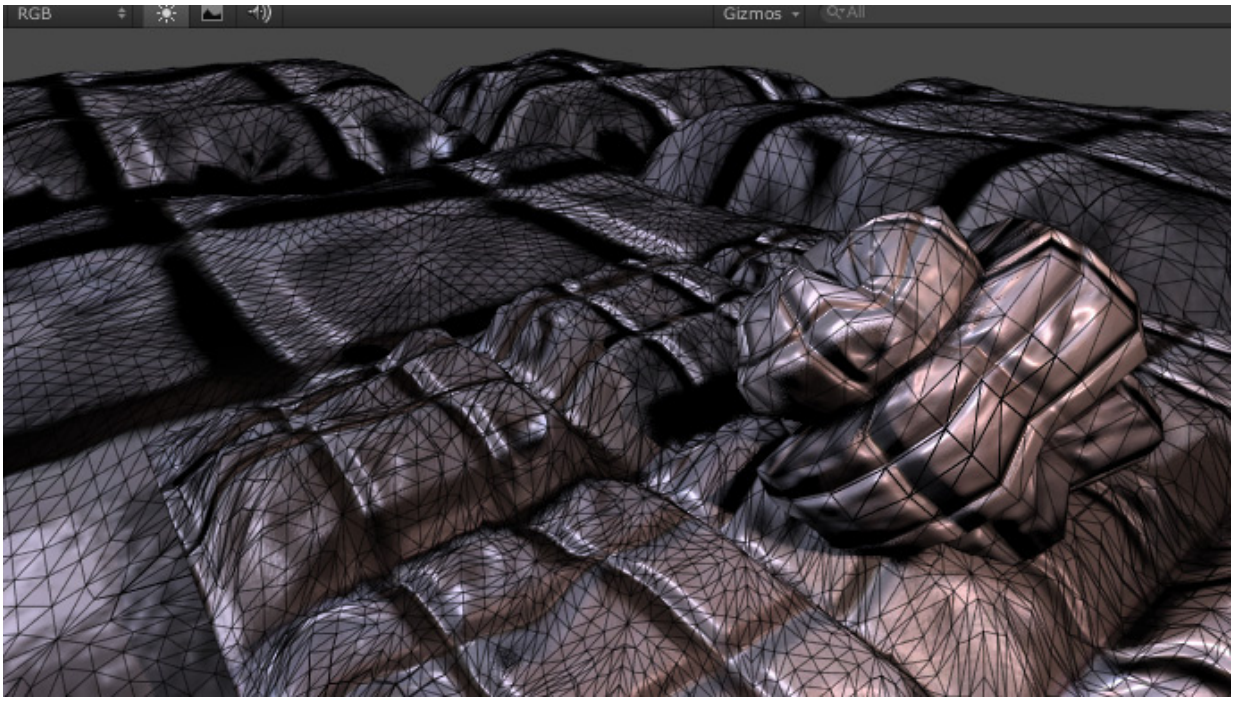
Purely distance based tessellation is effective only when triangle sizes are quite similar. In the image above, the GameObjects that have small triangles are tessellated too much, while GameObjects that have large triangles aren't tessellated enough.

One way to improve this is to compute tessellation levels based on triangle edge length on the screen. Unity should apply a larger tessellation factor to longer edges.

```
Shader "Tessellation Sample" {
    Properties {
        _EdgeLength ("Edge length", Range(2,50)) = 15
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DispTex ("Disp Texture", 2D) = "gray" {}
        _NormalMap ("Normalmap", 2D) = "bump" {}
        _Displacement ("Displacement", Range(0, 1.0)) = 0.3
        _Color ("Color", color) = (1,1,1,0)
        _SpecColor ("Spec color", color) = (0.5,0.5,0.5,0.5)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp tesse
        #pragma target 4.6
        #include "Tessellation.cginc"

        struct appdata {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
```

```
            };

            float _EdgeLength;

            float4 tessEdge (appdata v0, appdata v1, appdata v2)
            {
                return UnityEdgeLengthBasedTess (v0.vertex, v1.vertex, v2.vertex, _EdgeLen
            }

            sampler2D _DispTex;
            float _Displacement;

            void disp (inout appdata v)
            {
                float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
                v.vertex.xyz += v.normal * d;
            }

            struct Input {
                float2 uv_MainTex;
            };

            sampler2D _MainTex;
            sampler2D _NormalMap;
            fixed4 _Color;

            void surf (Input IN, inout SurfaceOutput o) {
                half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
                o.Albedo = c.rgb;
                o.Specular = 0.2;
                o.Gloss = 1.0;
                o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
            }
            ENDCG
        }
        FallBack "Diffuse"
    }
```

In this example, you call the `UnityEdgeLengthBasedTess` function from *Tessellation.cginc* to do all the work.



For performance reasons, call the UnityEdgeLengthBasedTessCull function instead, which performs patch frustum culling. This makes the shader a bit more expensive, but saves a lot of GPU work for parts of meshes that are outside of the Camera's view.

Phong Tessellation

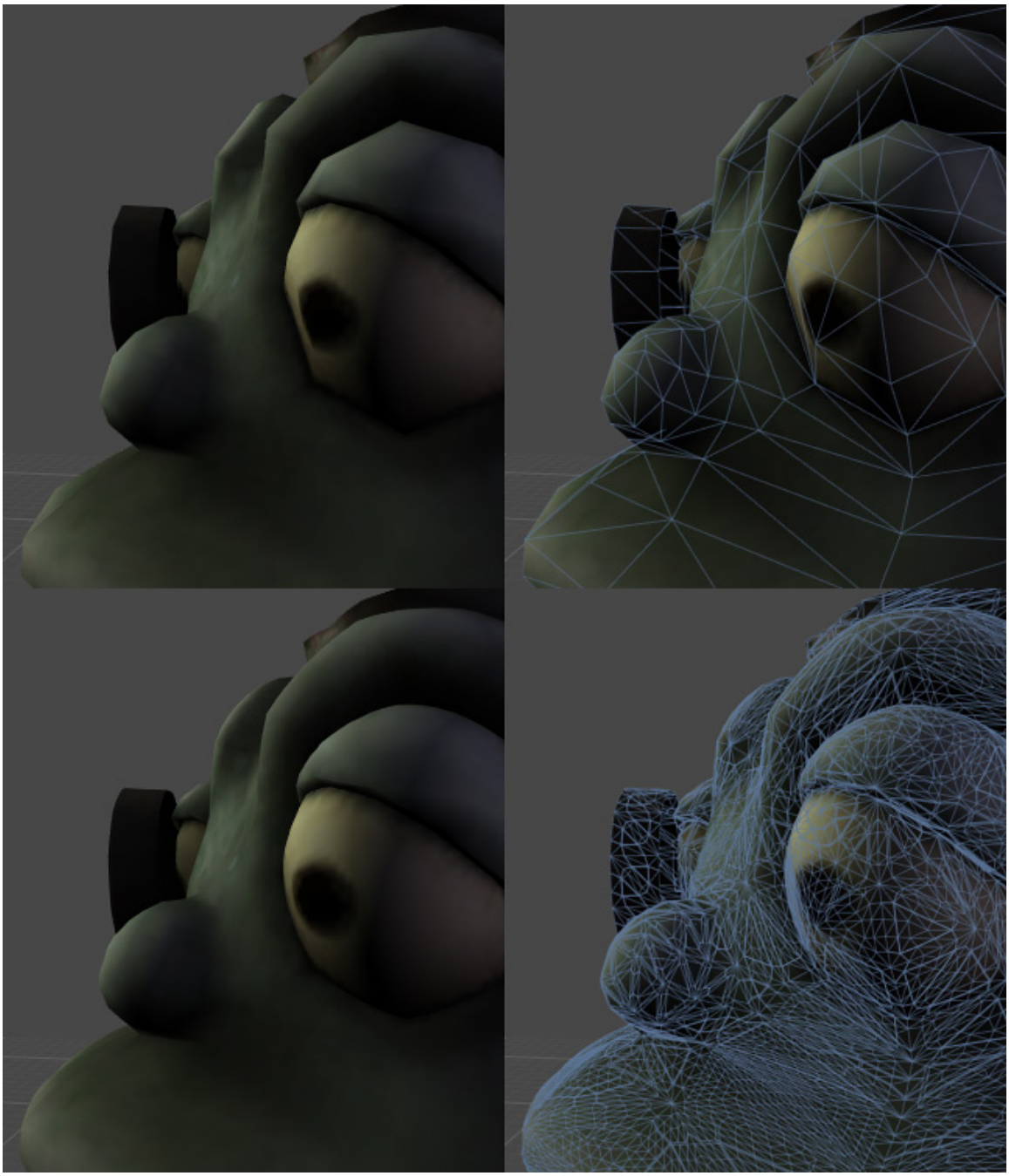
Phong Tessellation modifies positions of the subdivided faces so that the resulting surface follows the mesh normals a bit. It's quite an effective way of making low-poly meshes become more smooth.

Unity's surface shaders can compute Phong tessellation automatically using `tessphong:VariableName` compilation directive. Here's an example shader:

```
Shader "Phong Tessellation" {
    Properties {
        _EdgeLength ("Edge length", Range(2,50)) = 5
        _Phong ("Phong Strengh", Range(0,1)) = 0.5
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _Color ("Color", color) = (1,1,1,0)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf Lambert vertex:dispNone tessellate:tessEdge tessphong:_Ph
        #include "Tessellation.cginc"

        struct appdata {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float2 texcoord : TEXCOORD0;
        };

        void dispNone (inout appdata v) { }
```

```
        float _Phong;
        float _EdgeLength;

        float4 tessEdge (appdata v0, appdata v1, appdata v2)
        {
            return UnityEdgeLengthBasedTess (v0.vertex, v1.vertex, v2.vertex, _EdgeLen
        }

        struct Input {
            float2 uv_MainTex;
        };

        fixed4 _Color;
        sampler2D _MainTex;

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }

        ENDCG
    }
    FallBack "Diffuse"
}
```

Here is a comparison between a regular shader (top row) and one that uses Phong tessellation (bottom row). See that even without any displacement mapping, the surface becomes more round.



2018–03–20 Page amended with editorial review

Tessellation for Metal added in 2018.1

Writing vertex and fragment shaders

Leave feedback

ShaderLab shaders encompass more than just "hardware shaders". They do many things. They describe properties that are displayed in the Material **Inspector**, contain multiple **shader** implementations for different graphics hardware, configure fixed function hardware state and so on. The actual programmable shaders - like vertex and fragment programs - are just a part of the whole ShaderLab's "**shader**" concept. Take a look at shader tutorial for a basic introduction. Here we'll call the low-level hardware shaders **shader programs**.

If you want to write shaders that interact with lighting, take a look at Surface Shaders documentation. For some examples, take a look at **Vertex and Fragment Shader Examples** . The rest of this page assumes shaders do not interact with Unity lights (for example special effects, post-processed effects etc.)

Shader programs are written in HLSL language, by embedding "snippets" in the shader text, somewhere inside the Pass command. They usually look like this:

```
Pass {
    // ... the usual pass state setup ...

    CGPROGRAM
    // compilation directives for this snippet, e.g.:
    #pragma vertex vert
    #pragma fragment frag

    // the Cg/HLSL code itself

    ENDCG
    // ... the rest of pass setup ...
}
```

HLSL snippets

HLSL program snippets are written between **CGPROGRAM** and **ENDCG** keywords, or alternatively between **HLSLPROGRAM** and **ENDHLSL**. The latter form does *not* automatically include HLSLSupport and UnityShaderVariables built-in header files.

At the start of the snippet compilation directives can be given as **#pragma** statements. Directives indicating which shader functions to compile:

#pragma vertex *name* - compile function *name* as the **vertex shader**.
#pragma fragment *name* - compile function *name* as the **fragment shader**.
#pragma geometry *name* - compile function *name* as DX10 geometry shader. Having this option automatically turns on **#pragma target 4.0**, described below.
#pragma hull *name* - compile function *name* as DX11 hull shader. Having this option automatically turns on **#pragma target 5.0**, described below.

#pragma domain *name* - compile function *name* as DX11 domain shader. Having this option automatically turns on **#pragma target 5.0**, described below.
Other compilation directives:

#pragma target *name* - which shader target to compile to. See Shader Compilation Targets page for details.
#pragma require *feature* ... - fine grained control on which GPU features a shader needs, see Shader Compilation Targets page for details.
#pragma only_renderers *space separated names* - compile shader only for given renderers. By default shaders are compiled for all renderers. See *Renderers* below for details.
#pragma exclude_renderers *space separated names* - do not compile shader for given renderers. By default shaders are compiled for all renderers. See *Renderers* below for details.
#pragma multi_compile ... - for working with multiple shader variants.
#pragma enable_d3d11_debug_symbols - generate debug information for shaders compiled for DirectX 11, this will allow you to debug shaders via Visual Studio 2012 (or higher) Graphics debugger.
#pragma hardware_tier_variants *renderer name* - generate multiple shader hardware variants of each compiled shader, for each hardware tier that could run the selected renderer. See *Renderers* below for details.
Each snippet must contain at least a vertex program and a fragment program. Thus **#pragma vertex** and **#pragma fragment** directives are required.

Compilation directives that don't do anything starting with Unity 5.0 and can be safely removed: `#pragma glsl`, `#pragma glsl_no_auto_normalization`, `#pragma profileoption`, `#pragma fragmentoption`.

Unity only supports **#pragma** directives in the shader files, and not in the includes.

Rendering platforms

Unity supports several **rendering** APIs (e.g. Direct3D 11 and OpenGL), and by default all shader programs are compiled into all supported renderers. You can indicate which renderers to compile to using **#pragma only_renderers** or **#pragma exclude_renderers** directives. This is mostly useful in cases where you are explicitly using some shader language features that you know aren't possible on some platforms. Supported renderer names are:

d3d11 - Direct3D 11/12
glcore - OpenGL 3.x/4.x
gles - OpenGL ES 2.0
gles3 - OpenGL ES 3.x
metal - **iOS**/Mac Metal
vulkan - Vulkan
d3d11_9x - Direct3D 11 9.x feature level, as commonly used on WSA platforms
xboxone - Xbox One
ps4 - PlayStation 4
n3ds - Nintendo 3DS
wiiu - Nintendo Wii U
For example, this line would only compile shader into D3D11 mode:

```
#pragma only_renderers d3d11
```

See Also

Accessing Material Properties.
Writing Multiple Shader Program Variants.
Shader Compilation Targets.
Shading Language Details.
Shader Preprocessor Macros.
Platform Specific Rendering Differences.

2018–03–20 Page amended with editorial review

Shader #pragma directives added in Unity 2018.1

Vertex and fragment shader examples

Leave feedback

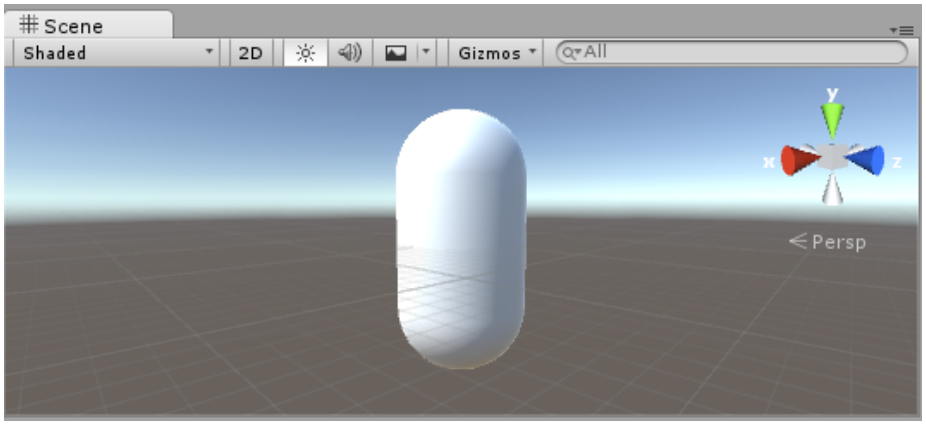
This page contains vertex and fragment program examples. For a basic introduction to shaders, see the shader tutorials: Part 1 and Part 2. For an easy way of writing regular material shaders, see Surface Shaders.

You can download the examples shown below as a zipped Unity project.

Setting up the scene

If you are not familiar with Unity's **Scene View**, **Hierarchy View**, **Project View** and **Inspector**, now would be a good time to read the first few sections from the manual, starting with Unity Basics.

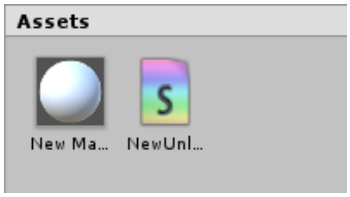
The first step is to create some objects which you will use to test your shaders. Select **Game Object** > **3D Object** > **Capsule** in the main menu. Then position the camera so it shows the capsule. Double-click the Capsule in the Hierarchy to focus the scene view on it, then select the Main Camera object and click **Game object** > **Align with View** from the main menu.



Create a new Material by selecting **Create** > **Material** from the menu in the Project View. A new material called *New Material* will appear in the Project View.

Creating a shader

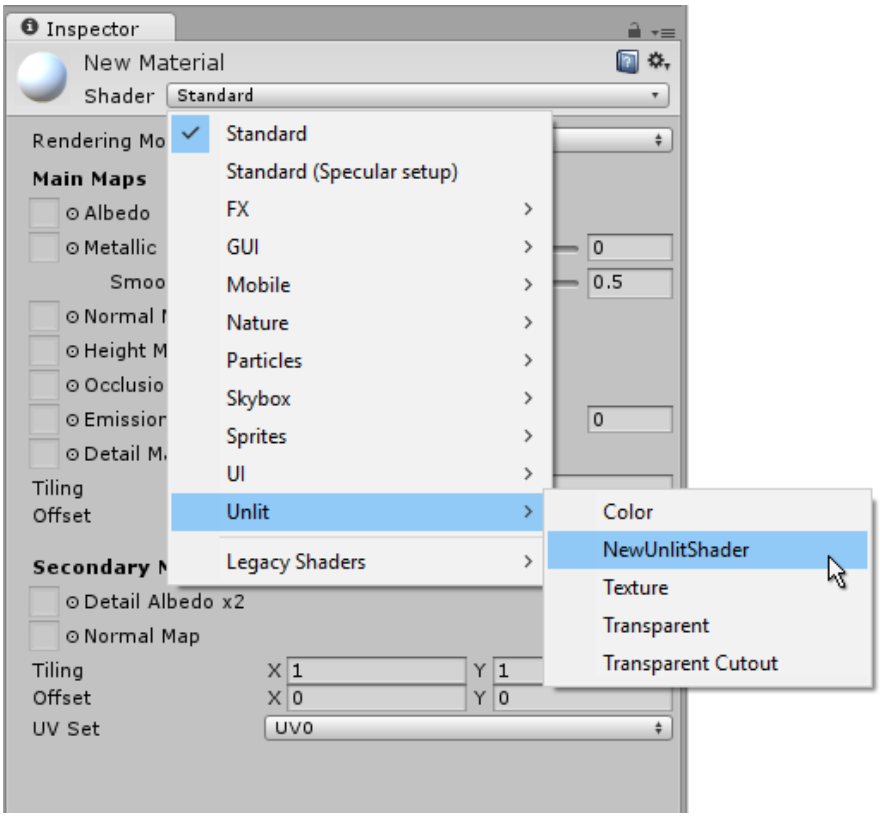
Now create a new Shader asset in a similar way. Select **Create** > **Shader** > **Unlit Shader** from the menu in the Project View. This creates a basic shader that just displays a texture without any lighting.



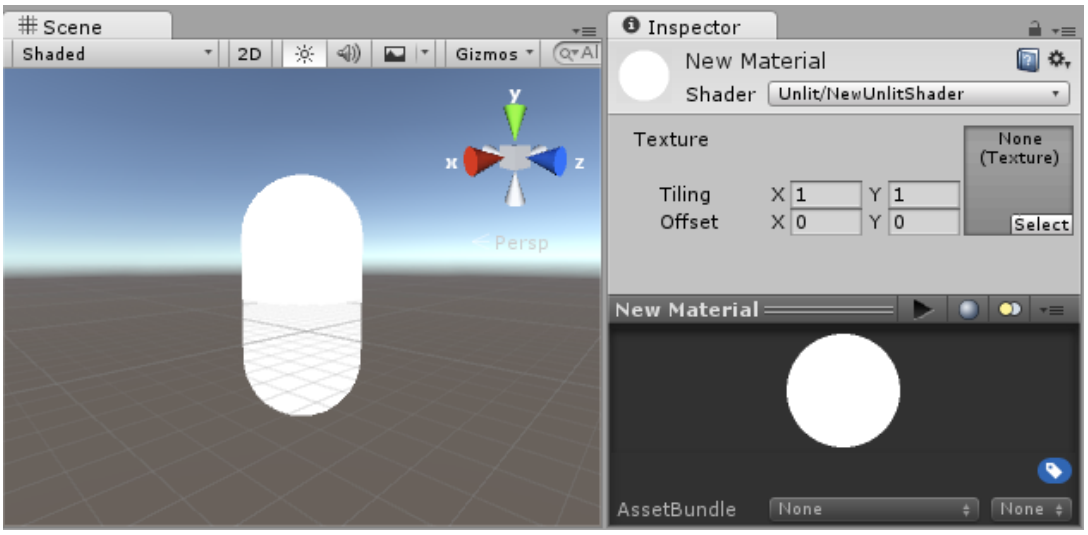
Other entries in the **Create** > **Shader** menu create barebone shaders or other types, for example a basic surface shader.

Linking the mesh, material and shader

Make the material use the shader via the material's inspector, or just drag the shader **asset** over the material **asset** in the Project View. The material inspector will display a white sphere when it uses this shader.



Now drag the material onto your **mesh** object in either the **Scene** or the Hierarchy views. Alternatively, select the object, and in the inspector make it use the material in the Mesh Renderer **component**'s Materials slot.



With these things set up, you can now begin looking at the shader code, and you will see the results of your changes to the shader on the capsule in the Scene View.

Main parts of the shader

To begin examining the code of the shader, double-click the shader asset in the Project View. The shader code will open in your script editor (MonoDevelop or Visual Studio).

The shader starts off with this code:

```
Shader "Unlit/NewUnlitShader"
{
    Properties
    {
```

```
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                UNITY_FOG_COORDS(1)
                float4 vertex : SV_POSITION;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                UNITY_TRANSFER_FOG(o,o.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                // sample the texture
                fixed4 col = tex2D(_MainTex, i.uv);
                // apply fog
                UNITY_APPLY_FOG(i.fogCoord, col);
                return col;
            }
            ENDCG
        }
    }
}
```

This initial shader does not look very simple! But don't worry, we will go over each part step-by-step.

Let's see the main parts of our simple shader.

```
Shader
```

The Shader command contains a string with the name of the shader. You can use forward slash characters "/" to place your shader in sub-menus when selecting your shader in the Material inspector.

```
Properties
```

The Properties block contains shader variables (textures, colors etc.) that will be saved as part of the Material, and displayed in the material inspector. In our unlit shader template, there is a single texture property declared.

```
SubShader
```

A Shader can contain one or more SubShaders, which are primarily used to implement shaders for different GPU capabilities. In this tutorial we're not much concerned with that, so all our shaders will contain just one SubShader.

```
Pass
```

Each SubShader is composed of a number of passes, and each Pass represents an execution of the vertex and fragment code for the same object rendered with the material of the shader. Many simple shaders use just one pass, but shaders that interact with lighting might need more (see Lighting Pipeline for details). Commands inside Pass typically setup fixed function state, for example blending modes.

```
__CGPROGRAM__ .. ENDCG
```

These keywords surround portions of HLSL code within the vertex and fragment shaders. Typically this is where most of the interesting code is. See vertex and fragment shaders for details.

Simple unlit shader

The unlit shader template does a few more things than would be absolutely needed to display an object with a texture. For example, it supports Fog, and texture tiling/offset fields in the material. Let's simplify the shader to bare minimum, and add more comments:

```
Shader "Unlit/SimpleUnlitTexturedShader"
{
    Properties
    {
        // we have removed support for texture tiling/offset,
        // so make them not be displayed in material inspector
        [NoScaleOffset] _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            // use "vert" function as the vertex shader
            #pragma vertex vert
            // use "frag" function as the pixel (fragment) shader
            #pragma fragment frag

            // vertex shader inputs
            struct appdata
            {
                float4 vertex : POSITION; // vertex position
                float2 uv : TEXCOORD0; // texture coordinate
            };

            // vertex shader outputs ("vertex to fragment")
            struct v2f
            {
                float2 uv : TEXCOORD0; // texture coordinate
                float4 vertex : SV_POSITION; // clip space position
            };

            // vertex shader
            v2f vert (appdata v)
            {
                v2f o;
                // transform position to clip space
                // (multiply with model*view*projection matrix)
                o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
                // just pass the texture coordinate
                o.uv = v.uv;
                return o;
            }

            // texture we will sample
            sampler2D _MainTex;

            // pixel shader; returns low precision ("fixed4" type)
            // color ("SV_Target" semantic)
            fixed4 frag (v2f i) : SV_Target
```

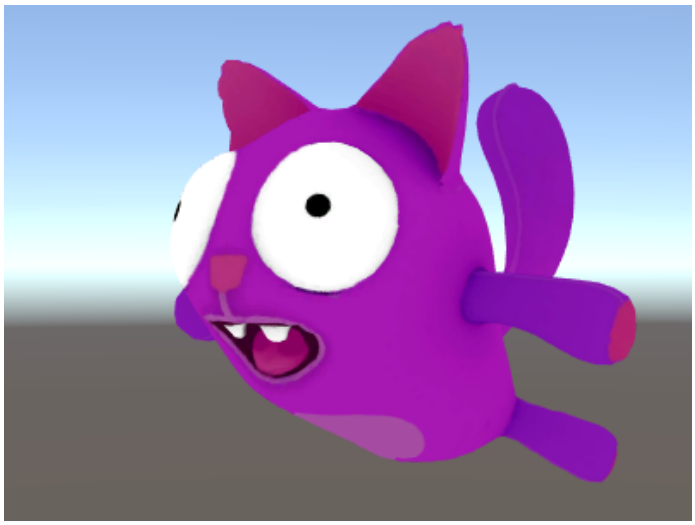
```
            {
                // sample texture and return it
                fixed4 col = tex2D(_MainTex, i.uv);
                return col;
            }
            ENDCG
        }
    }
}
```

The **Vertex Shader** is a program that runs on each vertex of the 3D model. Quite often it does not do anything particularly interesting. Here we just transform vertex position from object space into so called "clip space", which is what's used by the GPU to rasterize the object on screen. We also pass the input texture coordinate unmodified - we'll need it to sample the texture in the fragment shader.

The **Fragment Shader** is a program that runs on each and every pixel that object occupies on-screen, and is usually used to calculate and output the color of each pixel. Usually there are millions of **pixels** on the screen, and the fragment shaders are executed for all of them! Optimizing fragment shaders is quite an important part of overall game performance work.

Some variable or function definitions are followed by a **Semantic Signifier** - for example **: POSITION** or **: SV_Target**. These semantics signifiers communicate the "meaning" of these variables to the GPU. See the shader semantics page for details.

When used on a nice model with a nice texture, our simple shader looks pretty good!



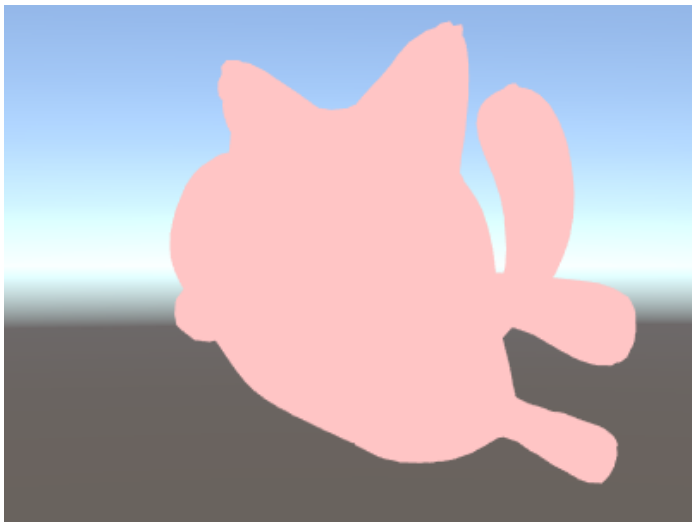
Even simpler single color shader

Let's simplify the shader even more – we'll make a shader that draws the whole object in a single color. This is not terribly useful, but hey we're learning here.

```
Shader "Unlit/SingleColor"
{
    Properties
    {
        // Color property for material inspector, default to white
        _Color ("Main Color", Color) = (1,1,1,1)
    }
    SubShader
```

```
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            // vertex shader
            // this time instead of using "appdata" struct, just spell inputs manually,
            // and instead of returning v2f struct, also just return a single output
            // float4 clip position
            float4 vert (float4 vertex : POSITION) : SV_POSITION
            {
                return mul(UNITY_MATRIX_MVP, vertex);
            }

            // color from the material
            fixed4 _Color;

            // pixel shader, no inputs needed
            fixed4 frag () : SV_Target
            {
                return _Color; // just return it
            }
            ENDCG
        }
    }
}
```

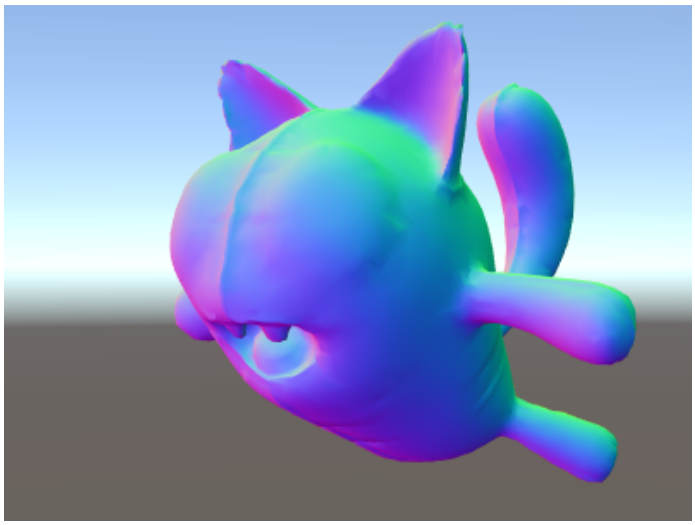
This time instead of using structs for input (**appdata**) and output (**v2f**), the shader functions just spell out inputs manually. Both ways work, and which you choose to use depends on your coding style and preference.



Using mesh normals for fun and profit

Let's proceed with a shader that displays mesh normals in world space. Without further ado:

```
Shader "Unlit/WorldSpaceNormals"
{
    // no Properties block this time!
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // include file that contains UnityObjectToWorldNormal helper function
            #include "UnityCG.cginc"

            struct v2f {
                // we'll output world space normal as one of regular ("texcoord") interpolator
                half3 worldNormal : TEXCOORD0;
                float4 pos : SV_POSITION;
            };

            // vertex shader: takes object space normal as input too
            v2f vert (float4 vertex : POSITION, float3 normal : NORMAL)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(vertex);
                // UnityCG.cginc file contains function to transform
                // normal from object to world space, use that
                o.worldNormal = UnityObjectToWorldNormal(normal);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 c = 0;
                // normal is a 3D vector with xyz components; in -1..1
                // range. To display it as color, bring the range into 0..1
                // and put into red, green, blue components
                c.rgb = i.worldNormal*0.5+0.5;
                return c;
            }
            ENDCG
        }
    }
}
```



Besides resulting in pretty colors, normals are used for all sorts of graphics effects – lighting, reflections, silhouettes and so on.

In the shader above, we started using one of Unity's built-in shader include files. Here, **UnityCG.cginc** was used which contains a handy function **UnityObjectToWorldNormal**. We have also used the utility function **UnityObjectToClipPos**, which transforms the vertex from object space to the screen. This just makes the code easier to read and is more efficient under certain circumstances.

We've seen that data can be passed from the vertex into fragment shader in so-called "interpolators" (or sometimes called "varyings"). In HLSL shading language they are typically labeled with **TEXCOORDn** semantic, and each of them can be up to a 4-component vector (see semantics page for details).

Also we've learned a simple technique in how to visualize normalized vectors (in –1.0 to +1.0 range) as colors: just multiply them by half and add half. See more vertex data visualization examples in vertex program inputs page.

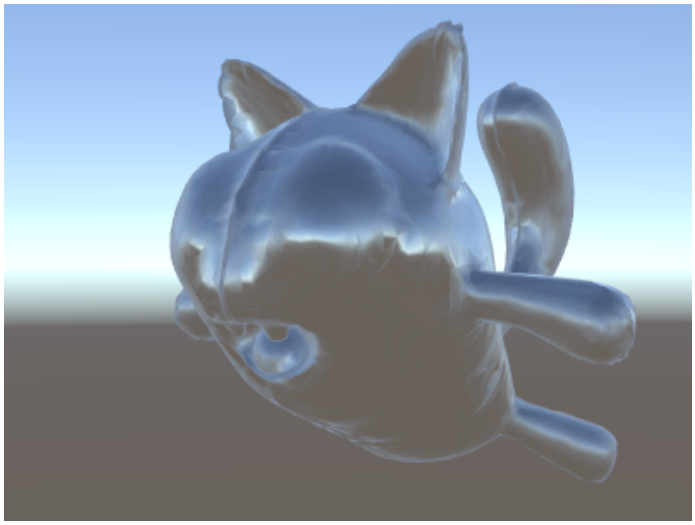
Environment reflection using world-space normals

When a Skybox is used in the scene as a reflection source (see Lighting Window), then essentially a "default" Reflection Probe is created, containing the skybox data. A reflection probe is internally a Cubemap texture; we will extend the world-space normals shader above to look into it.

The code is starting to get a bit involved by now. Of course, if you want shaders that automatically work with lights, shadows, reflections and the rest of the lighting system, it's way easier to use surface shaders. This example is intended to show you how to use parts of the lighting system in a "manual" way.

```
Shader "Unlit/SkyReflection"
{
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct v2f {
                half3 worldRefl : TEXCOORD0;
                float4 pos : SV_POSITION;
            };

            v2f vert (float4 vertex : POSITION, float3 normal : NORMAL)
            {
```

```
                v2f o;
                o.pos = UnityObjectToClipPos(vertex);
                // compute world space position of the vertex
                float3 worldPos = mul(_Object2World, vertex).xyz;
                // compute world space view direction
                float3 worldViewDir = normalize(UnityWorldSpaceViewDir(worldPos));
                // world space normal
                float3 worldNormal = UnityObjectToWorldNormal(normal);
                // world space reflection vector
                o.worldRefl = reflect(-worldViewDir, worldNormal);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                // sample the default reflection cubemap, using the reflection vector
                half4 skyData = UNITY_SAMPLE_TEXCUBE(unity_SpecCube0, i.worldRefl);
                // decode cubemap data into actual color
                half3 skyColor = DecodeHDR (skyData, unity_SpecCube0_HDR);
                // output it!
                fixed4 c = 0;
                c.rgb = skyColor;
                return c;
            }
            ENDCG
        }
    }
}
```



The example above uses several things from the built-in shader include files:

unity_SpecCube0, **unity_SpecCube0_HDR**, **Object2World**, **UNITY_MATRIX_MVP** from the built-in shader variables.
unity_SpecCube0 contains data for the active reflection probe.
UNITY_SAMPLE_TEXCUBE is a built-in macro to sample a cubemap. Most regular cubemaps are declared and used using standard HLSL syntax (**samplerCUBE** and **texCUBE**), however the reflection probe cubemaps in Unity are declared in a special way to save on sampler slots. If you don't know what that is, don't worry, just know that in order to use **unity_SpecCube0** cubemap you have to use **UNITY_SAMPLE_TEXCUBE** macro.

UnityWorldSpaceViewDir function from **UnityCG.cginc**, and **DecodeHDR** function from the same file. The latter is used to get actual color from the reflection probe data – since Unity stores reflection probe cubemap in specially encoded way.
reflect is just a built-in HLSL function to compute vector reflection around a given normal.

Environment reflection with a normal map

Often **Normal Maps** are used to create additional detail on objects, without creating additional geometry. Let's see how to make a shader that reflects the environment, with a normal map texture.

Now the math is starting to get *really involved*, so we'll do it in a few steps. In the shader above, the reflection direction was computed per-vertex (in the vertex shader), and the fragment shader was only doing the reflection probe cubemap lookup. However once we start using normal maps, the surface normal itself needs to be calculated on a per-pixel basis, which means we also have to compute how the environment is reflected per-pixel!

So first of all, let's rewrite the shader above to do the same thing, except we will move some of the calculations to the fragment shader, so they are computed per-pixel:

```
Shader "Unlit/SkyReflection Per Pixel"
{
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct v2f {
                float3 worldPos : TEXCOORD0;
                half3 worldNormal : TEXCOORD1;
                float4 pos : SV_POSITION;
            };

            v2f vert (float4 vertex : POSITION, float3 normal : NORMAL)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(vertex);
                o.worldPos = mul(_Object2World, vertex).xyz;
                o.worldNormal = UnityObjectToWorldNormal(normal);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                // compute view direction and reflection vector
                // per-pixel here
                half3 worldViewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
                half3 worldRefl = reflect(-worldViewDir, i.worldNormal);

                // same as in previous shader
                half4 skyData = UNITY_SAMPLE_TEXCUBE(unity_SpecCube0, worldRefl);
                half3 skyColor = DecodeHDR (skyData, unity_SpecCube0_HDR);
                fixed4 c = 0;
                c.rgb = skyColor;
```

```
                return c;
            }
            ENDCG
        }
    }
}
```

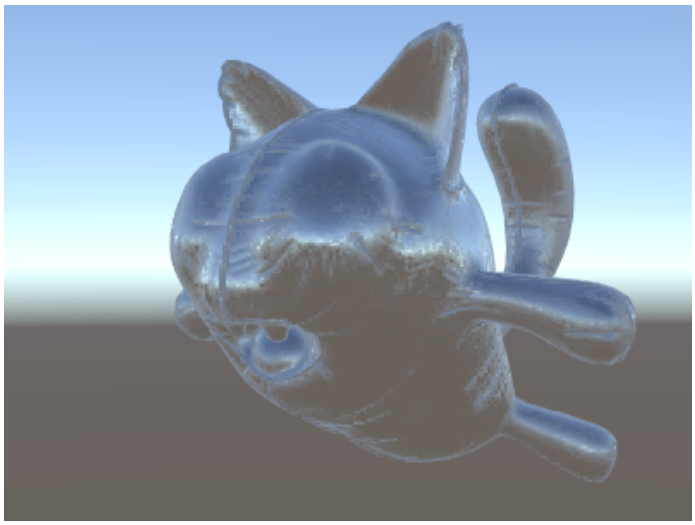
That by itself does not give us much – the shader looks exactly the same, except now it runs slower since it does more calculations for each and every pixel on screen, instead of only for each of the model's vertices. However, we'll need these calculations really soon. Higher graphics fidelity often requires more complex shaders.

We'll have to learn a new thing now too; the so-called "tangent space". Normal map textures are most often expressed in a coordinate space that can be thought of as "following the surface" of the model. In our shader, we will need to to know the tangent space basis vectors, read the normal vector from the texture, transform it into world space, and then do all the math from the above shader. Let's get to it!

```
Shader "Unlit/SkyReflection Per Pixel"
{
    Properties {
        // normal map texture on the material,
        // default to dummy "flat surface" normalmap
        _BumpMap("Normal Map", 2D) = "bump" {}
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct v2f {
                float3 worldPos : TEXCOORD0;
                // these three vectors will hold a 3x3 rotation matrix
                // that transforms from tangent to world space
                half3 tspace0 : TEXCOORD1; // tangent.x, bitangent.x, normal.x
                half3 tspace1 : TEXCOORD2; // tangent.y, bitangent.y, normal.y
                half3 tspace2 : TEXCOORD3; // tangent.z, bitangent.z, normal.z
                // texture coordinate for the normal map
                float2 uv : TEXCOORD4;
                float4 pos : SV_POSITION;
            };

            // vertex shader now also needs a per-vertex tangent vector.
            // in Unity tangents are 4D vectors, with the .w component used to
            // indicate direction of the bitangent vector.
            // we also need the texture coordinate.
            v2f vert (float4 vertex : POSITION, float3 normal : NORMAL, float4 tangent : TANGE
            {
                v2f o;
```

```
                o.pos = UnityObjectToClipPos(vertex);
                o.worldPos = mul(_Object2World, vertex).xyz;
                half3 wNormal = UnityObjectToWorldNormal(normal);
                half3 wTangent = UnityObjectToWorldDir(tangent.xyz);
                // compute bitangent from cross product of normal and tangent
                half tangentSign = tangent.w * unity_WorldTransformParams.w;
                half3 wBitangent = cross(wNormal, wTangent) * tangentSign;
                // output the tangent space matrix
                o.tspace0 = half3(wTangent.x, wBitangent.x, wNormal.x);
                o.tspace1 = half3(wTangent.y, wBitangent.y, wNormal.y);
                o.tspace2 = half3(wTangent.z, wBitangent.z, wNormal.z);
                o.uv = uv;
                return o;
            }

            // normal map texture from shader properties
            sampler2D _BumpMap;

            fixed4 frag (v2f i) : SV_Target
            {
                // sample the normal map, and decode from the Unity encoding
                half3 tnormal = UnpackNormal(tex2D(_BumpMap, i.uv));
                // transform normal from tangent to world space
                half3 worldNormal;
                worldNormal.x = dot(i.tspace0, tnormal);
                worldNormal.y = dot(i.tspace1, tnormal);
                worldNormal.z = dot(i.tspace2, tnormal);

                // rest the same as in previous shader
                half3 worldViewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
                half3 worldRefl = reflect(-worldViewDir, worldNormal);
                half4 skyData = UNITY_SAMPLE_TEXCUBE(unity_SpecCube0, worldRefl);
                half3 skyColor = DecodeHDR (skyData, unity_SpecCube0_HDR);
                fixed4 c = 0;
                c.rgb = skyColor;
                return c;
            }
            ENDCG
        }
    }
}
```

Phew, that was quite involved. But look, normal mapped reflections!



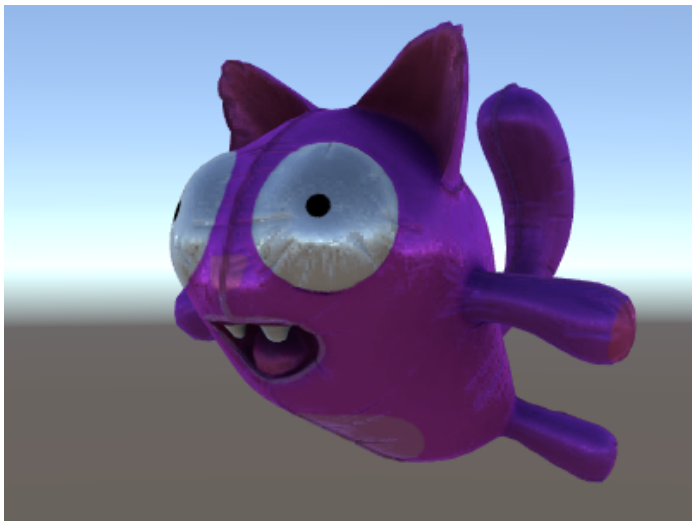
Adding more textures

Let's add more textures to the normal-mapped, sky-reflecting shader above. We'll add the base color texture, seen in the first unlit example, and an occlusion map to darken the cavities.

```
Shader "Unlit/More Textures"
{
    Properties {
        // three textures we'll use in the material
        _MainTex("Base texture", 2D) = "white" {}
        _OcclusionMap("Occlusion", 2D) = "white" {}
        _BumpMap("Normal Map", 2D) = "bump" {}
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            // exactly the same as in previous shader
            struct v2f {
                float3 worldPos : TEXCOORD0;
                half3 tspace0 : TEXCOORD1;
                half3 tspace1 : TEXCOORD2;
                half3 tspace2 : TEXCOORD3;
                float2 uv : TEXCOORD4;
                float4 pos : SV_POSITION;
            };
            v2f vert (float4 vertex : POSITION, float3 normal : NORMAL, float4 tangent : TANGE
            {
                v2f o;
                o.pos = UnityObjectToClipPos(vertex);
                o.worldPos = mul(_Object2World, vertex).xyz;
                half3 wNormal = UnityObjectToWorldNormal(normal);
                half3 wTangent = UnityObjectToWorldDir(tangent.xyz);
                half tangentSign = tangent.w * unity_WorldTransformParams.w;
```

```
                half3 wBitangent = cross(wNormal, wTangent) * tangentSign;
                o.tspace0 = half3(wTangent.x, wBitangent.x, wNormal.x);
                o.tspace1 = half3(wTangent.y, wBitangent.y, wNormal.y);
                o.tspace2 = half3(wTangent.z, wBitangent.z, wNormal.z);
                o.uv = uv;
                return o;
            }

            // textures from shader properties
            sampler2D _MainTex;
            sampler2D _OcclusionMap;
            sampler2D _BumpMap;

            fixed4 frag (v2f i) : SV_Target
            {
                // same as from previous shader...
                half3 tnormal = UnpackNormal(tex2D(_BumpMap, i.uv));
                half3 worldNormal;
                worldNormal.x = dot(i.tspace0, tnormal);
                worldNormal.y = dot(i.tspace1, tnormal);
                worldNormal.z = dot(i.tspace2, tnormal);
                half3 worldViewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
                half3 worldRefl = reflect(-worldViewDir, worldNormal);
                half4 skyData = UNITY_SAMPLE_TEXCUBE(unity_SpecCube0, worldRefl);
                half3 skyColor = DecodeHDR (skyData, unity_SpecCube0_HDR);
                fixed4 c = 0;
                c.rgb = skyColor;

                // modulate sky color with the base texture, and the occlusion map
                fixed3 baseColor = tex2D(_MainTex, i.uv).rgb;
                fixed occlusion = tex2D(_OcclusionMap, i.uv).r;
                c.rgb *= baseColor;
                c.rgb *= occlusion;

                return c;
            }
            ENDCG
        }
    }
}
```

Balloon cat is looking good!



Texturing shader examples

Procedural checkerboard pattern

Here's a shader that outputs a checkerboard pattern based on texture coordinates of a mesh:

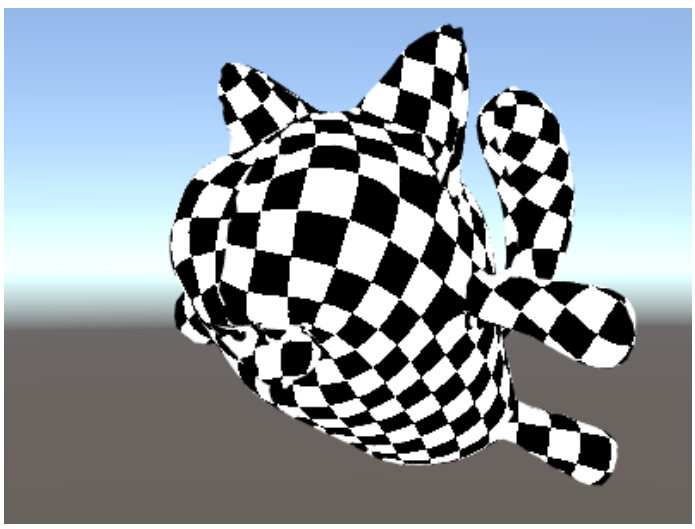
```
Shader "Unlit/Checkerboard"
{
    Properties
    {
        _Density ("Density", Range(2,50)) = 30
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
            };

            float _Density;

            v2f vert (float4 pos : POSITION, float2 uv : TEXCOORD0)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(pos);
                o.uv = uv * _Density;
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                float2 c = i.uv;
```

```
                c = floor(c) / 2;
                float checker = frac(c.x + c.y) * 2;
                return checker;
            }
            ENDCG
        }
    }
}
```

The density slider in the Properties block controls how dense the checkerboard is. In the vertex shader, the mesh UVs are multiplied by the density value to take them from a range of 0 to 1 to a range of 0 to density. Let's say the density was set to 30 - this will make **i.uv** input into the fragment shader contain floating point values from zero to 30 for various places of the mesh being rendered.

Then the fragment shader code takes only the integer part of the input coordinate using HLSL's built-in **floor** function, and divides it by two. Recall that the input coordinates were numbers from 0 to 30; this makes them all be "quantized" to values of 0, 0.5, 1, 1.5, 2, 2.5, and so on. This was done on both the x and y components of the input coordinate.

Next up, we add these x and y coordinates together (each of them only having possible values of 0, 0.5, 1, 1.5, ...) and only take the fractional part using another built-in HLSL function, **frac**. Result of this can only be either 0.0 or 0.5. We then multiply it by two to make it either 0.0 or 1.0, and output as a color (this results in black or white color respectively).

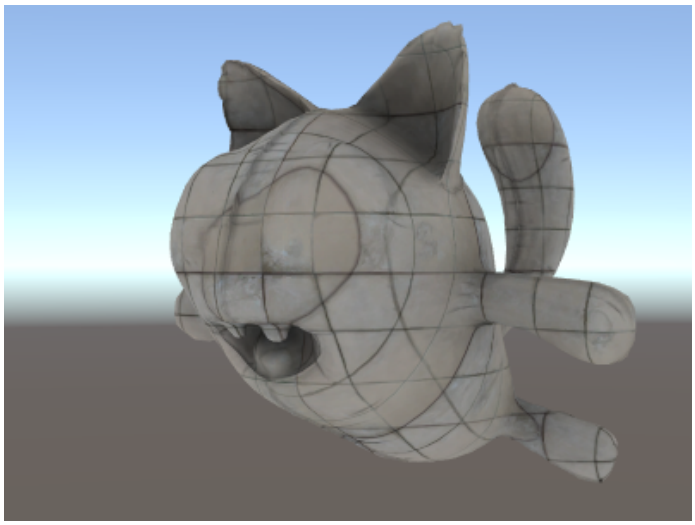


Tri-planar texturing

For complex or procedural meshes, instead of texturing them using the regular UV coordinates, it is sometimes useful to just "project" texture onto the object from three primary directions. This is called "tri-planar" texturing. The idea is to use surface normal to weight the three texture directions. Here's the shader:

```
Shader "Unlit/Triplanar"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Tiling ("Tiling", Float) = 1.0
        _OcclusionMap("Occlusion", 2D) = "white" {}
    }
    SubShader
    {
        Pass
```

```
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct v2f
            {
                half3 objNormal : TEXCOORD0;
                float3 coords : TEXCOORD1;
                float2 uv : TEXCOORD2;
                float4 pos : SV_POSITION;
            };

            float _Tiling;

            v2f vert (float4 pos : POSITION, float3 normal : NORMAL, float2 uv : TEXCOORD0)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(pos);
                o.coords = pos.xyz * _Tiling;
                o.objNormal = normal;
                o.uv = uv;
                return o;
            }

            sampler2D _MainTex;
            sampler2D _OcclusionMap;

            fixed4 frag (v2f i) : SV_Target
            {
                // use absolute value of normal as texture weights
                half3 blend = abs(i.objNormal);
                // make sure the weights sum up to 1 (divide by sum of x+y+z)
                blend /= dot(blend,1.0);
                // read the three texture projections, for x,y,z axes
                fixed4 cx = tex2D(_MainTex, i.coords.yz);
                fixed4 cy = tex2D(_MainTex, i.coords.xz);
                fixed4 cz = tex2D(_MainTex, i.coords.xy);
                // blend the textures based on weights
                fixed4 c = cx * blend.x + cy * blend.y + cz * blend.z;
                // modulate by regular occlusion map
                c *= tex2D(_OcclusionMap, i.uv);
                return c;
            }
            ENDCG
        }
    }
}
```



Calculating lighting

Typically when you want a shader that works with Unity's lighting pipeline, you would write a surface shader. This does most of the "heavy lifting" for you, and your shader code just needs to define surface properties.

However in some cases you want to bypass the standard surface shader path; either because you want to only support some limited subset of whole lighting pipeline for performance reasons, or you want to do custom things that aren't quite "standard lighting". The following examples will show how to get to the lighting data from manually-written vertex and fragment shaders. Looking at the code generated by surface shaders (via shader inspector) is also a good learning resource.

Simple diffuse lighting

The first thing we need to do is to indicate that our shader does in fact need lighting information passed to it. Unity's rendering pipeline supports various ways of **rendering**; here we'll be using the default forward rendering one.

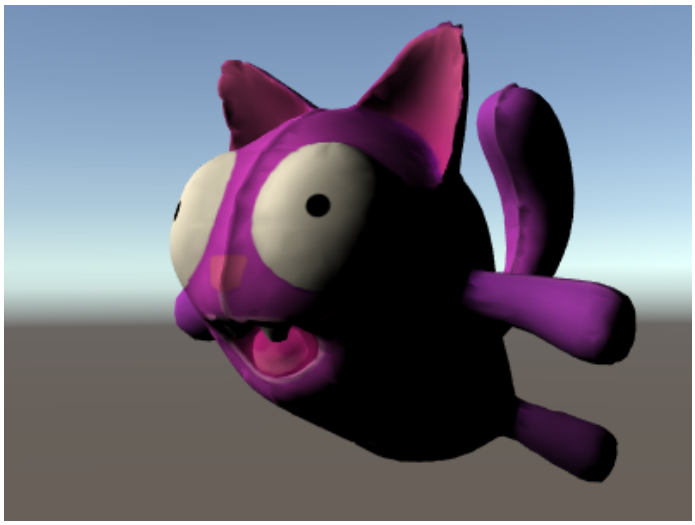
We'll start by only supporting one directional light. Forward rendering in Unity works by rendering the main directional light, ambient, **lightmaps** and reflections in a single pass called **ForwardBase**. In the shader, this is indicated by adding a pass tag: **Tags {"LightMode"="ForwardBase"}**. This will make directional light data be passed into shader via some built-in variables.

Here's the shader that computes simple diffuse lighting per vertex, and uses a single main texture:

```
Shader "Lit/Simple Diffuse"
{
    Properties
    {
        [NoScaleOffset] _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Pass
        {
            // indicate that our pass is the "base" pass in forward
            // rendering pipeline. It gets ambient and main directional
            // light data set up; light direction in _WorldSpaceLightPos0
            // and color in _LightColor0
            Tags {"LightMode"="ForwardBase"}

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc" // for UnityObjectToWorldNormal
```

```
            #include "UnityLightingCommon.cginc" // for _LightColor0

            struct v2f
            {
                float2 uv : TEXCOORD0;
                fixed4 diff : COLOR0; // diffuse lighting color
                float4 vertex : SV_POSITION;
            };

            v2f vert (appdata_base v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = v.texcoord;
                // get vertex normal in world space
                half3 worldNormal = UnityObjectToWorldNormal(v.normal);
                // dot product between normal and light direction for
                // standard diffuse (Lambert) lighting
                half nl = max(0, dot(worldNormal, _WorldSpaceLightPos0.xyz));
                // factor in the light color
                o.diff = nl * _LightColor0;
                return o;
            }

            sampler2D _MainTex;

            fixed4 frag (v2f i) : SV_Target
            {
                // sample texture
                fixed4 col = tex2D(_MainTex, i.uv);
                // multiply by lighting
                col *= i.diff;
                return col;
            }
            ENDCG
        }
    }
}
```

This makes the object react to light direction - parts of it facing the light are illuminated, and parts facing away are not illuminated at all.



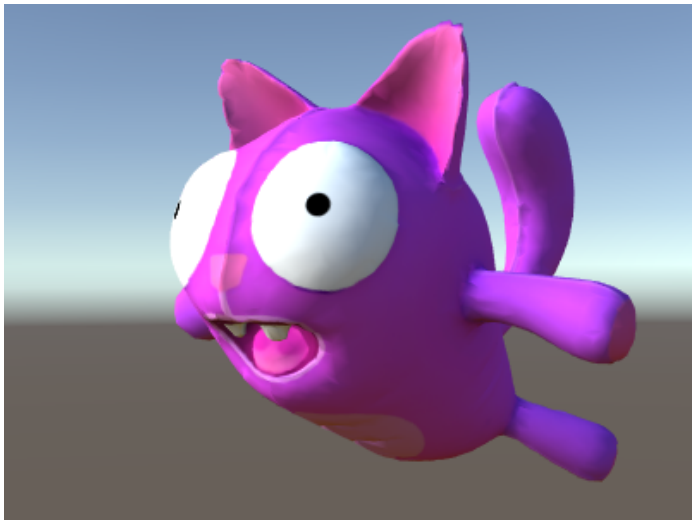
Diffuse lighting with ambient

The example above does not take any ambient lighting or light probes into account. Let's fix this! It turns out we can do this by adding just a single line of code. Both ambient and light probe data is passed to shaders in Spherical Harmonics form, and **ShadeSH9** function from **UnityCG.cginc** include file does all the work of evaluating it, given a world space normal.

```
Shader "Lit/Diffuse With Ambient"
{
    Properties
    {
        [NoScaleOffset] _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Pass
        {
            Tags {"LightMode"="ForwardBase"}

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"
            #include "UnityLightingCommon.cginc"

            struct v2f
            {
                float2 uv : TEXCOORD0;
                fixed4 diff : COLOR0;
                float4 vertex : SV_POSITION;
            };

            v2f vert (appdata_base v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = v.texcoord;
                half3 worldNormal = UnityObjectToWorldNormal(v.normal);
                half nl = max(0, dot(worldNormal, _WorldSpaceLightPos0.xyz));
                o.diff = nl * _LightColor0;
```

```
                // the only difference from previous shader:
                // in addition to the diffuse lighting from the main light,
                // add illumination from ambient or light probes
                // ShadeSH9 function from UnityCG.cginc evaluates it,
                // using world space normal
                o.diff.rgb += ShadeSH9(half4(worldNormal,1));
                return o;
            }

            sampler2D _MainTex;

            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 col = tex2D(_MainTex, i.uv);
                col *= i.diff;
                return col;
            }
            ENDCG
        }
    }
}
```

This shader is in fact starting to look very similar to the built-in Legacy Diffuse shader!



Implementing shadow casting

Our shader currently can neither receive nor cast shadows. Let's implement shadow casting first.

In order to cast shadows, a shader has to have a **ShadowCaster** pass type in any of its subshaders or any fallback. The ShadowCaster pass is used to render the object into the shadowmap, and typically it is fairly simple - the vertex shader only needs to evaluate the vertex position, and the fragment shader pretty much does not do anything. The shadowmap is only the depth buffer, so even the color output by the fragment shader does not really matter.

This means that for a lot of shaders, the shadow caster pass is going to be almost exactly the same (unless object has custom vertex shader based deformations, or has alpha cutout / semitransparent parts). The easiest way to pull it in is via UsePass shader command:

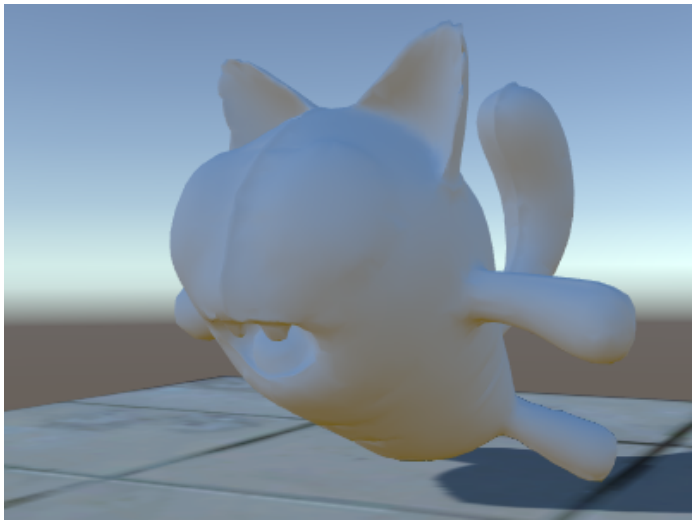
```
Pass
{
    // regular lighting pass
}
// pull in shadow caster from VertexLit built-in shader
UsePass "Legacy Shaders/VertexLit/SHADOWCASTER"
```

However we're learning here, so let's do the same thing "by hand" so to speak. For shorter code, we've replaced the lighting pass ("ForwardBase") with code that only does untextured ambient. Below it, there's a "ShadowCaster" pass that makes the object support shadow casting.

```
Shader "Lit/Shadow Casting"
{
    SubShader
    {
        // very simple lighting pass, that only does non-textured ambient
        Pass
        {
            Tags {"LightMode"="ForwardBase"}
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"
            struct v2f
            {
                fixed4 diff : COLOR0;
                float4 vertex : SV_POSITION;
            };
            v2f vert (appdata_base v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                half3 worldNormal = UnityObjectToWorldNormal(v.normal);
                // only evaluate ambient
                o.diff.rgb = ShadeSH9(half4(worldNormal,1));
                o.diff.a = 1;
                return o;
            }
            fixed4 frag (v2f i) : SV_Target
            {
                return i.diff;
            }
            ENDCG
        }

        // shadow caster rendering pass, implemented manually
        // using macros from UnityCG.cginc
        Pass
        {
            Tags {"LightMode"="ShadowCaster"}
```

```
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile_shadowcaster
            #include "UnityCG.cginc"

            struct v2f {
                V2F_SHADOW_CASTER;
            };

            v2f vert(appdata_base v)
            {
                v2f o;
                TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)
                return o;
            }

            float4 frag(v2f i) : SV_Target
            {
                SHADOW_CASTER_FRAGMENT(i)
            }
            ENDCG
        }
    }
}
```

Now there's a plane underneath, using a regular built-in **Diffuse shader**, so that we can see our shadows working (remember, our current shader does not support *receiving* shadows yet!).



We've used the **#pragma multi_compile_shadowcaster** directive. This causes the shader to be compiled into several variants with different preprocessor macros defined for each (see multiple shader variants page for details). When rendering into the shadowmap, the cases of point lights vs other light types need slightly different shader code, that's why this directive is needed.

Receiving shadows

Implementing support for receiving shadows will require compiling the base lighting pass into several variants, to handle cases of "directional light without shadows" and "directional light with shadows" properly. **#pragma multi_compile_fwdbase** directive does

this (see multiple shader variants for details). In fact it does a lot more: it also compiles variants for the different lightmap types, realtime GI being on or off **etc**. Currently we don't need all that, so we'll explicitly skip these variants.

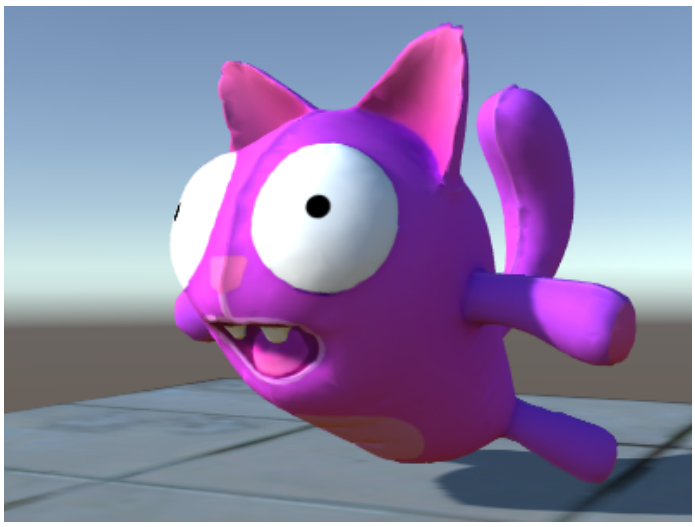
Then to get actual shadowing computations, we'll **#include "AutoLight.cginc"** shader include file and use SHADOW_COORDS, TRANSFER_SHADOW, SHADOW_ATTENUATION macros from it.

Here's the shader:

```
Shader "Lit/Diffuse With Shadows"
{
    Properties
    {
        [NoScaleOffset] _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Pass
        {
            Tags {"LightMode"="ForwardBase"}
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"
            #include "Lighting.cginc"

            // compile shader into multiple variants, with and without shadows
            // (we don't care about any lightmaps yet, so skip these variants)
            #pragma multi_compile_fwdbase nolightmap nodirlightmap nodynlightmap novertexlight
            // shadow helper functions and macros
            #include "AutoLight.cginc"

            struct v2f
            {
                float2 uv : TEXCOORD0;
                SHADOW_COORDS(1) // put shadows data into TEXCOORD1
                fixed3 diff : COLOR0;
                fixed3 ambient : COLOR1;
                float4 pos : SV_POSITION;
            };
            v2f vert (appdata_base v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv = v.texcoord;
                half3 worldNormal = UnityObjectToWorldNormal(v.normal);
                half nl = max(0, dot(worldNormal, _WorldSpaceLightPos0.xyz));
                o.diff = nl * _LightColor0.rgb;
                o.ambient = ShadeSH9(half4(worldNormal,1));
                // compute shadows data
                TRANSFER_SHADOW(o)
                return o;
            }

            sampler2D _MainTex;
```

```
            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 col = tex2D(_MainTex, i.uv);
                // compute shadow attenuation (1.0 = fully lit, 0.0 = fully shadowed)
                fixed shadow = SHADOW_ATTENUATION(i);
                // darken light's illumination with shadow, keep ambient intact
                fixed3 lighting = i.diff * shadow + i.ambient;
                col.rgb *= lighting;
                return col;
            }
            ENDCG
        }

        // shadow casting support
        UsePass "Legacy Shaders/VertexLit/SHADOWCASTER"
    }
}
```

Look, we have shadows now!



Other shader examples

Fog

```
Shader "Custom/TextureCoordinates/Fog" {
    SubShader {
        Pass {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            //Needed for fog variation to be compiled.
            #pragma multi_compile_fog

            #include "UnityCG.cginc"
```

```
            struct vertexInput {
                float4 vertex : POSITION;
                float4 texcoord0 : TEXCOORD0;
            };

            struct fragmentInput{
                float4 position : SV_POSITION;
                float4 texcoord0 : TEXCOORD0;

                //Used to pass fog amount around number should be a free texcoord.
                UNITY_FOG_COORDS(1)
            };

            fragmentInput vert(vertexInput i){
                fragmentInput o;
                o.position = UnityObjectToClipPos(i.vertex);
                o.texcoord0 = i.texcoord0;

                //Compute fog amount from clip space position.
                UNITY_TRANSFER_FOG(o,o.position);
                return o;
            }

            fixed4 frag(fragmentInput i) : SV_Target {
                fixed4 color = fixed4(i.texcoord0.xy,0,0);

                //Apply fog (additive pass are automatically handled)
                UNITY_APPLY_FOG(i.fogCoord, color);

                //to handle custom fog color another option would have been
                //#ifdef UNITY_PASS_FORWARDADD
                //  UNITY_APPLY_FOG_COLOR(i.fogCoord, color, float4(0,0,0,0));
                //#else
                //  fixed4 myCustomColor = fixed4(0,0,1,0);
                //  UNITY_APPLY_FOG_COLOR(i.fogCoord, color, myCustomColor);
                //#endif

                return color;
            }
            ENDCG
        }
    }
}
```

You can download the examples shown above as a zipped Unity project.

Further reading

Writing Vertex and Fragment Programs.
Shader Semantics.
Writing Surface Shaders.
Shader Reference.

Shader semantics

Leave feedback

When writing HLSL shader programs, input and output variables need to have their "intent" indicated via **semantics**. This is a standard concept in HLSL shader language; see the Semantics documentation on MSDN for more details.

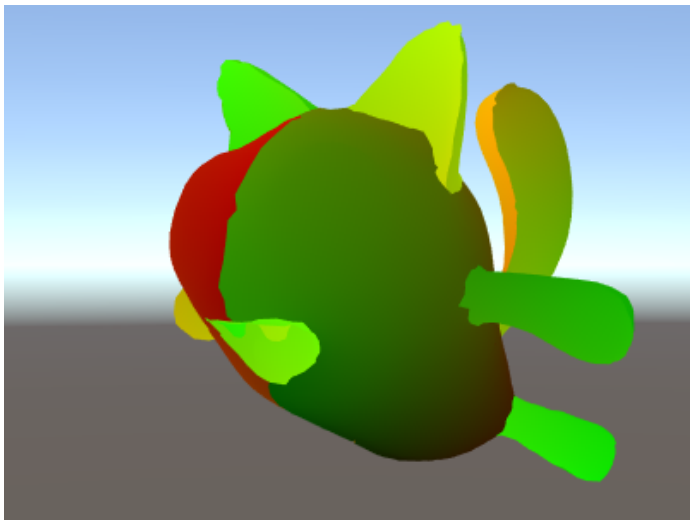
You can download the examples shown below as a zipped Unity project, here.

Vertex shader input semantics

The main **vertex shader** function (indicated by the `#pragma vertex` directive) needs to have semantics on all of the input parameters. These correspond to individual Mesh data elements, like vertex position, normal mesh, and texture coordinates. See vertex program inputs for more details.

Here's an example of a simple vertex **shader** that takes vertex position and a texture coordinate as an input. The **pixel** shader visualizes the texture coordinate as a color.

```
Shader "Unlit/Show UVs"
{
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            struct v2f {
                float2 uv : TEXCOORD0;
                float4 pos : SV_POSITION;
            };

            v2f vert (
                float4 vertex : POSITION, // vertex position input
                float2 uv : TEXCOORD0 // first texture coordinate input
                )
            {
                v2f o;
                o.pos = UnityObjectToClipPos(vertex);
                o.uv = uv;
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                return fixed4(i.uv, 0, 0);
            }
            ENDCG
        }
    }
}
```



Instead of spelling out all individual inputs one by one, it is also possible to declare a structure of them, and indicate semantics on each individual member variable of the struct. See shader program examples to learn how to do this.

Fragment shader output semantics

Most often a fragment (pixel) shader outputs a color, and has an `SV_Target` semantic. The fragment shader in the example above does exactly that:

```
fixed4 frag (v2f i) : SV_Target
```

The function `frag` has a return type of `fixed4` (low precision RGBA color). As it only returns a single value, the semantic is indicated on the function itself, `: SV_Target`.

It is also possible to return a structure with the outputs. The fragment shader above could be rewritten this way too, and it would do exactly the same:

```
struct fragOutput {
    fixed4 color : SV_Target;
};
fragOutput frag (v2f i)
{
    fragOutput o;
    o.color = fixed4(i.uv, 0, 0);
    return o;
}
```

Returning structures from the fragment shader is mostly useful for shaders that don't just return a single color. Additional semantics supported by the fragment shader outputs are as follows.

SV_TargetN: Multiple render targets

`SV_Target1`, `SV_Target2`, **etc.**: These are additional colors written by the shader. This is used when **rendering** into more than one render target at once (known as the Multiple Render Targets rendering technique, or MRT). `SV_Target0` is the same as `SV_Target`.

SV_Depth: Pixel shader depth output

Usually the fragment shader does not override the Z buffer value, and a default value is used from the regular triangle rasterization. However, for some effects it is useful to output custom Z buffer depth values per pixel.

Note that on many GPUs this turns off some depth buffer optimizations, so do not override Z buffer value without a good reason. The cost incurred by `SV_Depth` varies depending on the GPU architecture, but overall it's fairly similar to the cost of alpha testing (using the built-in `clip()` function in HLSL). Render shaders that modify depth after all regular opaque shaders (for example, by using the `AlphaTest` rendering queue.

The depth output value needs to be a single `float`.

Vertex shader outputs and fragment shader inputs

A vertex shader needs to output the final clip space position of a vertex, so that the GPU knows where on the screen to rasterize it, and at what depth. This output needs to have the `SV_POSITION` semantic, and be of a `float4` type.

Any other outputs ("interpolators" or "varyings") produced by the vertex shader are whatever your particular shader needs. The values output from the vertex shader will be interpolated across the face of the rendered triangles, and the values at each pixel will be passed as inputs to the **fragment shader**.

Many modern GPUs don't really care what semantics these variables have; however some old systems (most notably, shader model 2 GPUs on Direct3D 9) did have special rules about the semantics:

`TEXCOORD0`, `TEXCOORD1` etc are used to indicate arbitrary high precision data such as texture coordinates and positions.
`COLOR0` and `COLOR1` semantics on vertex outputs and fragment inputs are for low-precision, 0–1 range data (like simple color values).
For best cross platform support, label vertex outputs and fragment inputs as `TEXCOORDn` semantics.

See shader program examples for examples.

Interpolator count limits

There are limits to how many interpolator variables can be used in total to pass the information from the vertex into the fragment shader. The limit depends on the platform and GPU, and the general guidelines are:

Up to 8 interpolators: OpenGL ES 2.0 (iOS/Android), Direct3D 11 9.x level (Windows Phone) and Direct3 9 shader model 2.0 (old PCs). Since the interpolator count is limited, but each interpolator can be a 4-component vector, some shaders pack things together to stay within limits. For example, two texture coordinates can be passed in one `float4` variable (.xy for one coordinate, .zw for the second coordinate).
Up to 10 interpolators: Direct3D 9 shader model 3.0 (`#pragma target 3.0`).
Up to 16 interpolators: OpenGL ES 3.0 (iOS/Android), Metal (iOS).
Up to 32 interpolators: Direct3D 10 shader model 4.0 (`#pragma target 4.0`).
Regardless of your particular target hardware, it is generally a good idea to use as few interpolators as possible for performance reasons.

Other special semantics

Screen space pixel position: VPOS

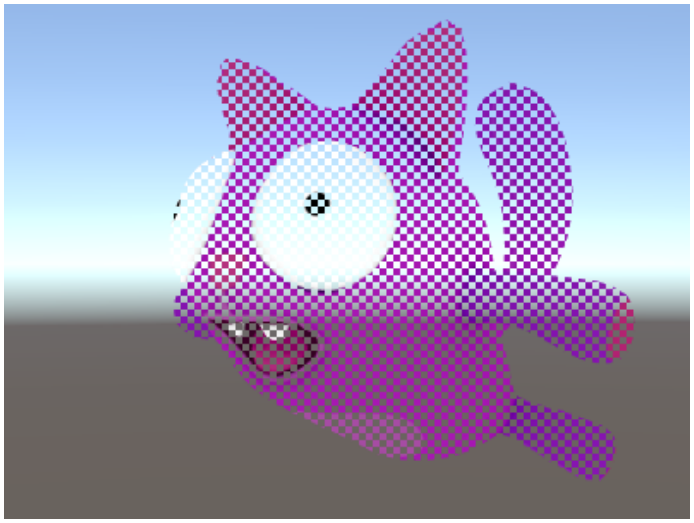
A fragment shader can receive position of the pixel being rendered as a special `VPOS` semantic. This feature only exists starting with shader model 3.0, so the shader needs to have the `#pragma target 3.0` compilation directive.

On different platforms the underlying type of the screen space position input varies, so for maximum portability use the `UNITY_VPOS_TYPE` type for it (it will be `float4` on most platforms, and float2 on Direct3D 9).

Additionally, using the pixel position semantic makes it hard to have both the clip space position (SV_POSITION) and VPOS in the same vertex-to-fragment structure. So the vertex shader should output the clip space position as a separate "out" variable. See the example shader below:

```
Shader "Unlit/Screen Position"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma target 3.0

            // note: no SV_POSITION in this struct
            struct v2f {
                float2 uv : TEXCOORD0;
            };

            v2f vert (
                float4 vertex : POSITION, // vertex position input
                float2 uv : TEXCOORD0, // texture coordinate input
                out float4 outpos : SV_POSITION // clip space position output
                )
            {
                v2f o;
                o.uv = uv;
                outpos = UnityObjectToClipPos(vertex);
                return o;
            }

            sampler2D _MainTex;

            fixed4 frag (v2f i, UNITY_VPOS_TYPE screenPos : VPOS) : SV_Target
            {
                // screenPos.xy will contain pixel integer coordinates.
                // use them to implement a checkerboard pattern that skips rendering
                // 4x4 blocks of pixels

                // checker value will be negative for 4x4 blocks of pixels
                // in a checkerboard pattern
                screenPos.xy = floor(screenPos.xy * 0.25) * 0.5;
                float checker = -frac(screenPos.r + screenPos.g);

                // clip HLSL instruction stops rendering a pixel if value is negative
                clip(checker);

                // for pixels that were kept, read the texture and output it
                fixed4 c = tex2D (_MainTex, i.uv);
                return c;
            }
            ENDCG
```

```
        }
    }
}
```



Face orientation: VFACE

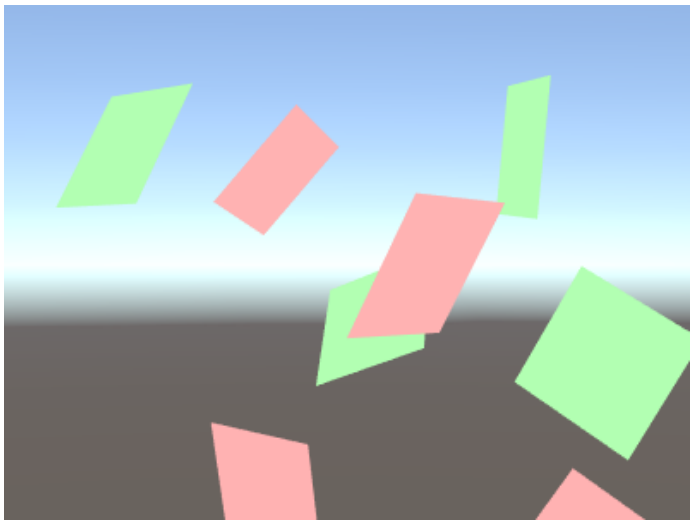
A fragment shader can receive a variable that indicates whether the rendered surface is facing the **camera**, or facing away from the **camera**. This is useful when rendering geometry that should be visible from both sides – often used on leaves and similar thin objects. The `VFACE` semantic input variable will contain a positive value for front-facing triangles, and a negative value for back-facing ones.

This feature only exists from shader model 3.0 onwards, so the shader needs to have the `#pragma target 3.0` compilation directive.

```
Shader "Unlit/Face Orientation"
{
    Properties
    {
        _ColorFront ("Front Color", Color) = (1,0.7,0.7,1)
        _ColorBack ("Back Color", Color) = (0.7,1,0.7,1)
    }
    SubShader
    {
        Pass
        {
            Cull Off // turn off backface culling

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma target 3.0

            float4 vert (float4 vertex : POSITION) : SV_POSITION
            {
                return UnityObjectToClipPos(vertex);
            }
```

```
            fixed4 _ColorFront;
            fixed4 _ColorBack;

            fixed4 frag (fixed facing : VFACE) : SV_Target
            {
                // VFACE input positive for frontbaces,
                // negative for backfaces. Output one
                // of the two colors depending on that.
                return facing > 0 ? _ColorFront : _ColorBack;
            }
            ENDCG
        }
    }
}
```

The shader above uses the Cull state to turn off backface culling (by default back-facing triangles are not rendered at all). Here is the shader applied to a bunch of **Quad** meshes, rotated at different orientations:



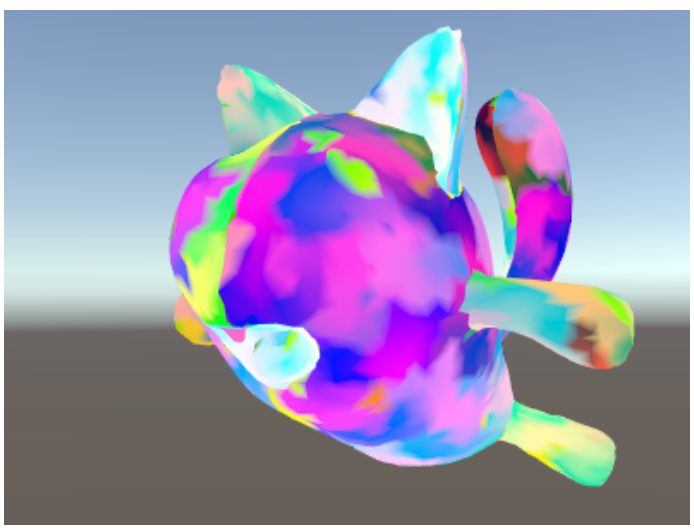
Vertex ID: SV_VertexID

A vertex shader can receive a variable that has the "vertex number" as an unsigned integer. This is mostly useful when you want to fetch additional per-vertex data from textures or ComputeBuffers.

This feature only exists from DX10 (shader model 4.0) and GLCore / OpenGL ES 3, so the shader needs to have the `#pragma target 3.5` compilation directive.

```
Shader "Unlit/VertexID"
{
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma target 3.5

            struct v2f {
```

```
            fixed4 color : TEXCOORD0;
            float4 pos : SV_POSITION;
        };

        v2f vert (
            float4 vertex : POSITION, // vertex position input
            uint vid : SV_VertexID // vertex ID, needs to be uint
            )
        {
            v2f o;
            o.pos = UnityObjectToClipPos(vertex);
            // output funky colors based on vertex ID
            float f = (float)vid;
            o.color = half4(sin(f/10),sin(f/100),sin(f/1000),0) * 0.5 + 0.5;
            return o;
        }

        fixed4 frag (v2f i) : SV_Target
        {
            return i.color;
        }
        ENDCG
    }
  }
}
```



(You can download the examples shown above as a zipped Unity project, here)

Accessing shader properties in Cg/HLSL

Leave feedback

Shader declares Material properties in a Properties block. If you want to access some of those properties in a shader program, you need to declare a Cg/HLSL variable with the same name and a matching type. An example is provided in Shader Tutorial: Vertex and Fragment Programs.

For example these **shader** properties:

```
_MyColor ("Some Color", Color) = (1,1,1,1)
_MyVector ("Some Vector", Vector) = (0,0,0,0)
_MyFloat ("My float", Float) = 0.5
_MyTexture ("Texture", 2D) = "white" {}
_MyCubemap ("Cubemap", CUBE) = "" {}
```

would be declared for access in Cg/HLSL code as:

```
fixed4 _MyColor; // low precision type is usually enough for colors
float4 _MyVector;
float _MyFloat;
sampler2D _MyTexture;
samplerCUBE _MyCubemap;
```

Cg/HLSL can also accept **uniform** keyword, but it is not necessary:

```
uniform float4 _MyColor;
```

Property types in **ShaderLab** map to Cg/HLSL variable types this way:

Color and Vector properties map to **float4**, **half4** or **fixed4** variables.
Range and Float properties map to **float**, **half** or **fixed** variables.
Texture properties map to **sampler2D** variables for regular (2D) textures; **Cubemaps** map to **samplerCUBE**; and 3D textures map to **sampler3D**.

How property values are provided to shaders

Shader property values are found and provided to shaders from these places:

Per-Renderer values set in MaterialPropertyBlock. This is typically “per-instance” data (e.g. customized tint color for a lot of objects that all share the same material).
Values set in the Material that’s used on the rendered object.

Global shader properties, set either by Unity **rendering** code itself (see built-in shader variables), or from your own **scripts** (e.g. Shader.SetGlobalTexture).

The order of precedence is like above: per-instance data overrides everything; then Material data is used; and finally if shader property does not exist in these two places then global property value is used. Finally, if there's no shader property value defined anywhere, then "default" (zero for floats, black for colors, empty white texture for textures) value will be provided.

Serialized and Runtime Material properties

Materials can contain both serialized and runtime-set property values.

Serialized data is all the properties defined in shader's Properties block. Typically these are values that need to be stored in the material, and are tweakable by the user in Material **Inspector**.

A material can also have some properties that are used by the shader, but not declared in shader's Properties block. Typically this is for properties that are set from script code at runtime, e.g. via Material.SetColor. Note that matrices and arrays can only exist as non-serialized runtime properties (since there's no way to define them in Properties block).

Special Texture properties

For each texture that is setup as a shader/material property, Unity also sets up some extra information in additional vector properties.

Texture tiling & offset

Materials often have Tiling and Offset fields for their texture properties. This information is passed into shaders in a float4 *{TextureName}* `_ST` property:

`x` contains X tiling value
`y` contains Y tiling value
`z` contains X offset value
`w` contains Y offset value

For example, if a shader contains texture named `_MainTex`, the tiling information will be in a `_MainTex_ST` vector.

Texture size

{TextureName} `_TexelSize` - a float4 property contains texture size information:

`x` contains 1.0/width
`y` contains 1.0/height
`z` contains width
`w` contains height

Texture HDR parameters

{TextureName} `_HDR` - a float4 property with information on how to decode a potentially **HDR** (e.g. RGBM-encoded) texture depending on the color space used. See `DecodeHDR` function in UnityCG.cginc shader include file.

Color spaces and color/vector shader data

When using Linear color space, all material color properties are supplied as sRGB colors, but are converted into linear values when passed into shaders.

For example, if your Properties shader block contains a `Color` property called "*MyColor", then the corresponding* "MyColor" HLSL variable will get the linear color value.

For properties that are marked as `Float` or `Vector` type, no color space conversions are done by default; it is assumed that they contain non-color data. It is possible to add `[Gamma]` attribute for float/vector properties to indicate that they are specified in sRGB space, just like colors (see Properties).

See Also

ShaderLab Properties block.
Writing Shader Programs.

Providing vertex data to vertex programs

Leave feedback

For Cg/HLSL vertex programs, the Mesh vertex data is passed as inputs to the vertex shader function. Each input needs to have semantic speficied for it: for example, `POSITION` input is the vertex position, and `NORMAL` is the vertex normal.

Often, vertex data inputs are declared in a structure, instead of listing them one by one. Several commonly used vertex structures are defined in UnityCG.cginc include file, and in most cases it's enough just to use those. The structures are:

`appdata_base`: position, normal and one texture coordinate.
`appdata_tan`: position, tangent, normal and one texture coordinate.
`appdata_full`: position, tangent, normal, four texture coordinates and color.

Example: This **shader** colors the mesh based on its normals, and uses `appdata_base` as vertex program input:

```
Shader "VertexInputSimple" {
    SubShader {
        Pass {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct v2f {
                float4 pos : SV_POSITION;
                fixed4 color : COLOR;
            };

            v2f vert (appdata_base v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.color.xyz = v.normal * 0.5 + 0.5;
                o.color.w = 1.0;
                return o;
            }

            fixed4 frag (v2f i) : SV_Target { return i.color; }
            ENDCG
        }
    }
}
```

To access different vertex data, you need to declare the vertex structure yourself, or add input parameters to the vertex shader. Vertex data is identified by Cg/HLSL semantics, and must be from the following list:

`POSITION` is the vertex position, typically a `float3` or `float4`.
`NORMAL` is the vertex normal, typically a `float3`.
`TEXCOORD0` is the first UV coordinate, typically `float2`, `float3` or `float4`.
`TEXCOORD1`, `TEXCOORD2` and `TEXCOORD3` are the 2nd, 3rd and 4th UV coordinates, respectively.
`TANGENT` is the tangent vector (used for normal mapping), typically a `float4`.
`COLOR` is the per-vertex color, typically a `float4`.
When the mesh data contains fewer **components** than are needed by the vertex shader input, the rest are filled with zeroes, except for the `.w` component which defaults to 1. For example, mesh texture coordinates are often 2D vectors with just x and y components. If a vertex shader declares a `float4` input with `TEXCOORD0` semantic, the value received by the **vertex shader** will contain (x,y,0,1).

Examples

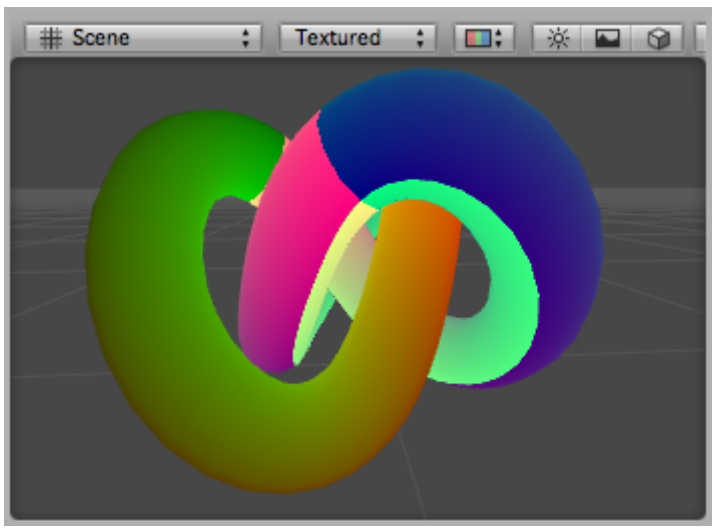
Visualizing UVs

The following shader example uses the vertex position and the first texture coordinate as the vertex shader inputs (defined in the structure **appdata**). This shader is very useful for debugging the UV coordinates of the mesh.

```
Shader "Debug/UV 1" {
SubShader {
    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        // vertex input: position, UV
        struct appdata {
            float4 vertex : POSITION;
            float4 texcoord : TEXCOORD0;
        };

        struct v2f {
            float4 pos : SV_POSITION;
            float4 uv : TEXCOORD0;
        };

        v2f vert (appdata v) {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex );
            o.uv = float4( v.texcoord.xy, 0, 0 );
            return o;
```

```
        }

        half4 frag( v2f i ) : SV_Target {
            half4 c = frac( i.uv );
            if (any(saturate(i.uv) - i.uv))
                c.b = 0.5;
            return c;
        }
        ENDCG
    }
}
}
```

Here, UV coordinates are visualized as red and green colors, while an additional blue tint has been applied to coordinates outside of the 0 to 1 range:



Debug UV1 shader applied to a torus knot model

Similarly, this shader vizualizes the second UV set of the model:

```
Shader "Debug/UV 2" {
SubShader {
    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        // vertex input: position, second UV
        struct appdata {
            float4 vertex : POSITION;
            float4 texcoord1 : TEXCOORD1;
```

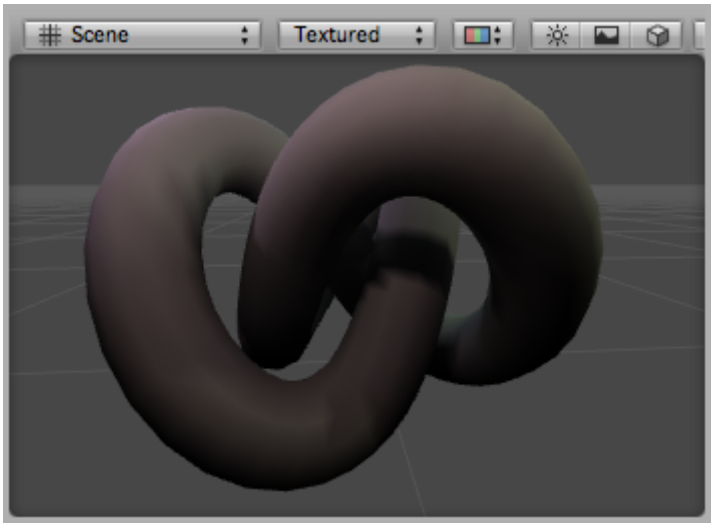
```
        };

        struct v2f {
            float4 pos : SV_POSITION;
            float4 uv : TEXCOORD0;
        };

        v2f vert (appdata v) {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex );
            o.uv = float4( v.texcoord1.xy, 0, 0 );
            return o;
        }

        half4 frag( v2f i ) : SV_Target {
            half4 c = frac( i.uv );
            if (any(saturate(i.uv) - i.uv))
                c.b = 0.5;
            return c;
        }
        ENDCG
    }
}
}
```

Visualizing vertex colors

The following shader uses the vertex position and the per-vertex colors as the vertex shader inputs (defined in structure **appdata**).

```
Shader "Debug/Vertex color" {
SubShader {
    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        // vertex input: position, color
        struct appdata {
            float4 vertex : POSITION;
            fixed4 color : COLOR;
        };
```

```
        struct v2f {
            float4 pos : SV_POSITION;
            fixed4 color : COLOR;
        };

        v2f vert (appdata v) {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex );
            o.color = v.color;
            return o;
        }

        fixed4 frag (v2f i) : SV_Target { return i.color; }
        ENDCG
    }
}
}
```



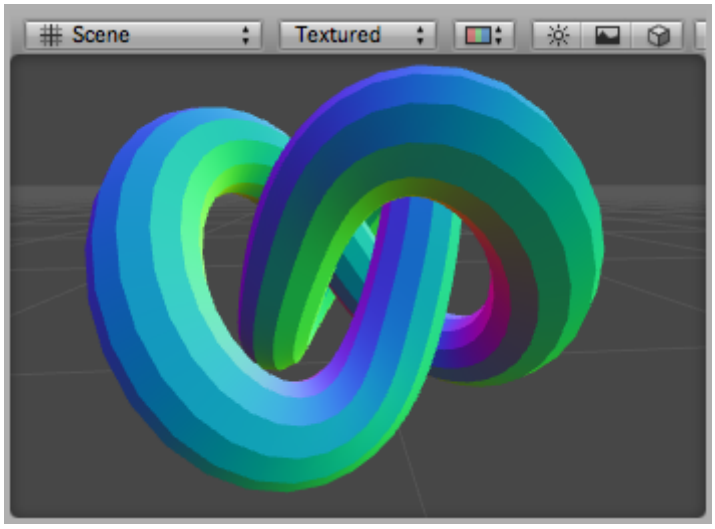
Debug Colors shader applied to a torus knot model that has illumination baked into colors

Visualizing normals

The following shader uses the vertex position and the normal as the vertex shader inputs (defined in the structure **appdata**). The normal's X,Y & Z components are visualized as RGB colors. Because the normal components are in the –1 to 1 range, we scale and bias them so that the output colors are displayable in the 0 to 1 range.

```
Shader "Debug/Normals" {
SubShader {
    Pass {
        CGPROGRAM
```

```
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        // vertex input: position, normal
        struct appdata {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
        };

        struct v2f {
            float4 pos : SV_POSITION;
            fixed4 color : COLOR;
        };

        v2f vert (appdata v) {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex );
            o.color.xyz = v.normal * 0.5 + 0.5;
            o.color.w = 1.0;
            return o;
        }

        fixed4 frag (v2f i) : SV_Target { return i.color; }
        ENDCG
    }
}
}
```



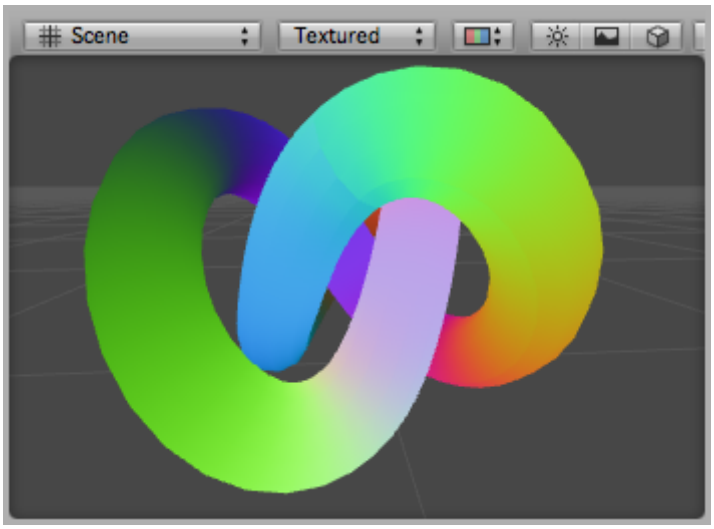
Debug Normals shader applied to a torus knot model. You can see that the model has hard shading edges.

Visualizing tangents and binormals

Tangent and binormal vectors are used for normal mapping. In Unity only the tangent vector is stored in vertices, and the binormal is derived from the normal and tangent values.

The following shader uses the vertex position and the tangent as vertex shader inputs (defined in structure **appdata**). Tangent's x,y and z components are visualized as RGB colors. Because the normal components are in the –1 to 1 range, we scale and bias them so that the output colors are in a displayable 0 to 1 range.

```
Shader "Debug/Tangents" {
SubShader {
    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        // vertex input: position, tangent
        struct appdata {
            float4 vertex : POSITION;
            float4 tangent : TANGENT;
        };

        struct v2f {
            float4 pos : SV_POSITION;
            fixed4 color : COLOR;
        };

        v2f vert (appdata v) {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex );
            o.color = v.tangent * 0.5 + 0.5;
            return o;
        }

        fixed4 frag (v2f i) : SV_Target { return i.color; }
        ENDCG
    }
}
}
```

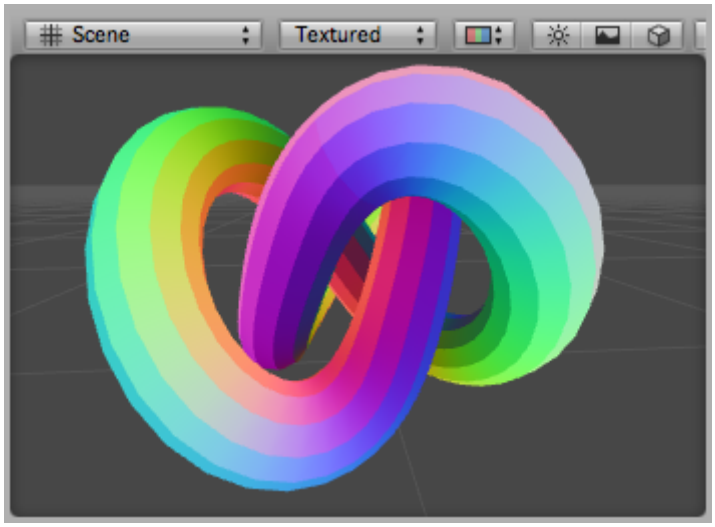


Debug Tangents shader applied to a torus knot model.

The following shader visualizes bitangents. It uses the vertex position, normal and tangent values as vertex inputs. The bitangent (sometimes called binormal) is calculated from the normal and tangent values. It needs to be scaled and biased into a displayable 0 to 1 range.

```
Shader "Debug/Bitangents" {
SubShader {
    Pass {
        Fog { Mode Off }
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        // vertex input: position, normal, tangent
        struct appdata {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float4 tangent : TANGENT;
        };

        struct v2f {
            float4 pos : SV_POSITION;
            float4 color : COLOR;
        };

        v2f vert (appdata v) {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex );
            // calculate bitangent
            float3 bitangent = cross( v.normal, v.tangent.xyz ) * v.tangent.w;
            o.color.xyz = bitangent * 0.5 + 0.5;
            o.color.w = 1.0;
```

```
            return o;
        }

        fixed4 frag (v2f i) : SV_Target { return i.color; }
        ENDCG
    }
}
}
```



Debug Bitangents shader applied to a torus knot model.

Further Reading

Shader Semantics
Vertex and Fragment Program Examples
Built-in Shader Include Files

Built-in shader include files

Leave feedback

Unity contains several files that can be used by your shader programs to bring in predefined variables and helper functions. This is done by the standard `#include` directive, e.g.:

```
CGPROGRAM
// ...
#include "UnityCG.cginc"
// ...
ENDCG
```

Shader include files in Unity are with `.cginc` extension, and the built-in ones are:

`HLSLSupport.cginc` - *(automatically included)* Helper macros and definitions for cross-platform **shader** compilation.
`UnityShaderVariables.cginc` - *(automatically included)* Commonly used global variables.
`UnityCG.cginc` - commonly used helper functions.
`AutoLight.cginc` - lighting & shadowing functionality, e.g. surface shaders use this file internally.
`Lighting.cginc` - standard surface shader lighting models; automatically included when you're writing surface shaders.
`TerrainEngine.cginc` - helper functions for **Terrain** & Vegetation shaders.
These files are found inside Unity application (**{unity install path}/Data/CGIncludes/UnityCG.cginc** on Windows, **/Applications/Unity/Unity.app/Contents/CGIncludes/UnityCG.cginc** on Mac), if you want to take a look at what exactly is done in any of the helper code.

HLSLSupport.cginc

This file is automatically included when compiling **CGPROGRAM** shaders (but not included for HLSLPROGRAM ones). It declares various preprocessor macros to aid in multi-platform shader development.

UnityShaderVariables.cginc

This file is automatically included when compiling CGPROGRAM shaders (but not included for HLSLPROGRAM ones). It declares various built-in global variables that are commonly used in shaders.

UnityCG.cginc

This file is often included in Unity shaders. It declares many built-in helper functions and data structures.

Data structures in UnityCG.cginc

struct `appdata_base`: **vertex shader** input with position, normal, one texture coordinate.
struct `appdata_tan`: vertex shader input with position, normal, tangent, one texture coordinate.
struct `appdata_full`: vertex shader input with position, normal, tangent, vertex color and two texture coordinates.

struct `appdata_img`: vertex shader input with position and one texture coordinate.

Predefined Shader preprocessor macros

Leave feedback

Unity defines several preprocessor macros when compiling Shader programs.

Target platform

Macro:	Target platform:
`SHADER_API_D3D11`	Direct3D 11
`SHADER_API_GLCORE`	Desktop OpenGL "core" (GL 3/4)
`SHADER_API_GLES`	OpenGL ES 2.0
`SHADER_API_GLES3`	OpenGL ES 3.0/3.1
`SHADER_API_METAL`	**iOS**/Mac Metal
`SHADER_API_VULKAN`	Vulkan
`SHADER_API_D3D11_9X`	Direct3D 11 "feature level 9.x" target for **Universal Windows Platform**
`SHADER_API_PS4`	PlayStation 4. `SHADER_API_PSSL` is also defined.
`SHADER_API_XBOXONE`	**Xbox One**

`SHADER_API_MOBILE` is defined for all general mobile platforms (GLES, GLES3, METAL).

Additionally, `SHADER_TARGET_GLSL` is defined when the target shading language is GLSL (always true for OpenGL/GLES platforms).

Shader target model

`SHADER_TARGET` is defined to a numeric value that matches the **Shader** target compilation model (that is, matching `#pragma target` directive). For example, `SHADER_TARGET` is `30` when compiling into Shader model 3.0. You can use it in Shader code to do conditional checks. For example:

```
#if SHADER_TARGET < 30
    // less than Shader model 3.0:
    // very limited Shader capabilities, do some approximation
#else
    // decent capabilities, do a better thing
#endif
```

Unity version

`UNITY_VERSION` contains the numeric value of the Unity version. For example, `UNITY_VERSION` is `501` for Unity 5.0.1. This can be used for version comparisons if you need to write Shaders that use different built-in Shader functionality. For example, a `#if UNITY_VERSION >= 500` preprocessor check only passes on versions 5.0.0 or later.

Shader stage being compiled

Preprocessor macros `SHADER_STAGE_VERTEX`, `SHADER_STAGE_FRAGMENT`, `SHADER_STAGE_DOMAIN`, `SHADER_STAGE_HULL`, `SHADER_STAGE_GEOMETRY`, `SHADER_STAGE_COMPUTE` are defined when compiling each Shader stage. Typically they are useful when sharing Shader code between **pixel** Shaders and compute Shaders, to handle cases where some things have to be done slightly differently.

Platform difference helpers

Direct use of these platform macros is discouraged, as they don't always contribute to the future-proofing of your code. For example, if you're writing a Shader that checks for D3D11, you may want to ensure that, in the future, the check is extended to include Vulkan. Instead, Unity defines several helper macros (in `HLSLSupport.cginc`):

Macro:	Use:
`UNITY_BRANCH`	Add this before conditional statements to tell the compiler that this should be compiled into an actual branch. Expands to `[branch]` when on HLSL platforms.
`UNITY_FLATTEN`	Add this before conditional statements to tell the compiler that this should be flattened to avoid an actual branch instruction. Expands to `[flatten]` when on HLSL platforms.
`UNITY_NO_SCREENSPACE_SHADOWS`	Defined on platforms that do not use cascaded screenspace shadowmaps (mobile platforms).
`UNITY_NO_LINEAR_COLORSPACE`	Defined on platforms that do not support Linear color space (mobile platforms).
`UNITY_NO_RGBM`	Defined on platforms where RGBM **compression** for **lightmaps** is not used (mobile platforms).
`UNITY_NO_DXT5nm`	Defined on platforms that do not use DXT5nm normal-map compression (mobile platforms).
`UNITY_FRAMEBUFFER_FETCH_AVAILABLE`	Defined on platforms where "framebuffer color fetch" functionality can be available (generally iOS platforms - OpenGL ES 2.0, 3.0 and Metal).
`UNITY_USE_RGBA_FOR_POINT_SHADOWS`	Defined on platforms where point light shadowmaps use RGBA Textures with encoded depth (other platforms use single-channel floating point Textures).
`UNITY_ATTEN_CHANNEL`	Defines which channel of light attenuation Texture contains the data; used in per-pixel lighting code. Defined to either 'r' or 'a'.
`UNITY_HALF_TEXEL_OFFSET`	Defined on platforms that need a half-texel offset adjustment in mapping texels to pixels (e.g. Direct3D 9).
`UNITY_UV_STARTS_AT_TOP`	Always defined with value of 1 or 0. A value of 1 is on platforms where Texture V coordinate is 0 at the "top" of the Texture. Direct3D-like platforms use value of 1; OpenGL-like platforms use value of 0.
`UNITY_MIGHT_NOT_HAVE_DEPTH_Texture`	Defined if a platform might emulate shadow maps or depth Textures by manually **rendering** depth into a Texture.
`UNITY_PROJ_COORD(a)`	Given a 4-component vector, this returns a Texture coordinate suitable for projected Texture reads. On most platforms this returns the given value directly.
`UNITY_NEAR_CLIP_VALUE`	Defined to the value of near **clipping plane**. Direct3D-like platforms use 0.0 while OpenGL-like platforms use –1.0.
`UNITY_VPOS_TYPE`	Defines the data type required for pixel position input (VPOS): `float2` on D3D9, `float4` elsewhere.
`UNITY_CAN_COMPILE_TESSELLATION`	Defined when the Shader compiler "understands" the tessellation Shader HLSL syntax (currently only D3D11).
`UNITY_INITIALIZE_OUTPUT(type,name)`	Initializes the variable *name* of given *type* to zero.

Macro:	Use:
`UNITY_COMPILER_HLSL`, `UNITY_COMPILER_HLSL2GLSL`, `UNITY_COMPILER_CG`	Indicates which Shader compiler is being used to compile Shaders - respectively: Microsoft's HLSL, HLSL to GLSL translator, and NVIDIA's Cg. See documentation on Shading Languages for more details. Use this if you run into very specific Shader syntax handling differences between the compilers, and want to write different code for each compiler.

`UNITY_REVERSED_Z` - defined on plaftorms using reverse Z buffer. Stored Z values are in the range 1..0 instead of 0..1.

Shadow mapping macros

Declaring and sampling shadow maps can be very different depending on the platform. Unity has several macros to help with this:

Macro:	Use:
`UNITY_DECLARE_SHADOWMAP(tex)`	Declares a shadowmap Texture variable with name "tex".
`UNITY_SAMPLE_SHADOW(tex,uv)`	Samples shadowmap Texture "tex" at given "uv" coordinate (XY components are Texture location, Z component is depth to compare with). Returns single float value with the shadow term in 0..1 range.
`UNITY_SAMPLE_SHADOW_PROJ(tex,uv)`	Similar to above, but does a projective shadowmap read. "uv" is a float4, all other **components** are divided by .w for doing the lookup.

NOTE: Not all graphics cards support shadowmaps. Use SystemInfo.SupportsRenderTextureFormat to check for support.

Constant buffer macros

Direct3D 11 groups all Shader variables into "constant buffers". Most of Unity's built-in variables are already grouped, but for variables in your own Shaders it might be more optimal to put them into separate constant buffers depending on expected frequency of updates.

Use `CBUFFER_START(name)` and `CBUFFER_END` macros for that:

```
CBUFFER_START(MyRarelyUpdatedVariables)
    float4 _SomeGlobalValue;
CBUFFER_END
```

Texture/Sampler declaration macros

Usually you would use `texture2D` in Shader code to declare a Texture and Sampler pair. However on some platforms (such as DX11), Textures and Samplers are separate **GameObjects**, and maximum possible Sampler count is quite limited. Unity has some macros to declare Textures without Samplers, and to sample a Texture using a Sampler from another Texture. Use this if you end up running into Sampler limits, and you know that several of your Textures can in fact share a Sampler (Samplers define Texture filtering and wrapping modes).

Macro:	Use:
`UNITY_DECLARE_TEX2D(name)`	Declares a Texture and Sampler pair.
`UNITY_DECLARE_TEX2D_NOSAMPLER(name)`	Declares a Texture without a Sampler.

Macro:	Use:
`UNITY_DECLARE_TEX2DARRAY(name)`	Declares a Texture array Sampler variable.
`UNITY_SAMPLE_TEX2D(name,uv)`	Sample from a Texture and Sampler pair, using given Texture coordinate.
`UNITY_SAMPLE_TEX2D_SAMPLER(name,samplername,uv)`	Sample from Texture (name), using a Sampler from another Texture (samplername).
`UNITY_SAMPLE_TEX2DARRAY(name,uv)`	Sample from a Texture array with a float3 UV; the z component of the coordinate is array element index.
`UNITY_SAMPLE_TEX2DARRAY_LOD(name,uv,lod)`	Sample from a Texture array with an explicit mipmap level.

For more information, see documentation on Sampler States.

Surface Shader pass indicators

When Surface Shaders are compiled, they generate a lot of code for various passes to do lighting. When compiling each pass, one of the following macros is defined:

Macro:	Use:
`UNITY_PASS_FORWARDBASE`	**Forward rendering** base pass (main directional light, lightmaps, SH).
`UNITY_PASS_FORWARDADD`	Forward rendering additive pass (one light per pass).
`UNITY_PASS_DEFERRED`	**Deferred shading** pass (renders g buffer).
`UNITY_PASS_SHADOWCASTER`	Shadow caster and depth Texture rendering pass.
`UNITY_PASS_PREPASSBASE`	Legacy deferred lighting base pass (renders normals and specular exponent).
`UNITY_PASS_PREPASSFINAL`	Legacy deferred lighting final pass (applies lighting and Textures).

Disable Auto-Upgrade

`UNITY_SHADER_NO_UPGRADE` allows you to disable Unity from automatically upgrading or modifying your shader file.

See also

Built-in Shader include files
Built-in Shader variables
Vertex and Fragment program examples

• 2017–05–16 Page amended with no editorial review

Built-in shader helper functions

Leave feedback

Unity has a number of built-in utility functions designed to make writing **shaders** simpler and easier.

Functions declared in UnityCG.cginc

See Built-in shader include files for an overview of shader include files provided with Unity.

Vertex transformation functions in UnityCG.cginc

Function:	Description:
`float4 UnityObjectToClipPos(float3 pos)`	Transforms a point from object space to the **camera**'s clip space in homogeneous coordinates. This is the equivalent of **mul(UNITY_MATRIX_MVP, float4(pos, 1.0))**, and should be used in its place.
`float3 UnityObjectToViewPos(float3 pos)`	Transforms a point from object space to view space. This is the equivalent of **mul(UNITY_MATRIX_MV, float4(pos, 1.0)).xyz**, and should be used in its place.

Generic helper functions in UnityCG.cginc

Function:	Description:
`float3 WorldSpaceViewDir (float4 v)`	Returns world space direction (not normalized) from given object space vertex position towards the camera.
`float3 ObjSpaceViewDir (float4 v)`	Returns object space direction (not normalized) from given object space vertex position towards the camera.
`float2 ParallaxOffset (half h, half height, half3 viewDir)`	calculates UV offset for parallax normal mapping.
`fixed Luminance (fixed3 c)`	Converts color to luminance (grayscale).
`fixed3 DecodeLightmap (fixed4 color)`	Decodes color from Unity **lightmap** (RGBM or dLDR depending on platform).
`float4 EncodeFloatRGBA (float v)`	Encodes [0..1) range float into RGBA color, for storage in low precision render target.
`float DecodeFloatRGBA (float4 enc)`	Decodes RGBA color into a float.
`float2 EncodeFloatRG (float v)`	Encodes [0..1) range float into a float2.
`float DecodeFloatRG (float2 enc)`	Decodes a previously-encoded RG float.
`float2 EncodeViewNormalStereo (float3 n)`	Encodes view space normal into two numbers in 0..1 range.
`float3 DecodeViewNormalStereo (float4 enc4)`	Decodes view space normal from enc4.xy.

Forward rendering helper functions in UnityCG.cginc

These functions are only useful when using **forward rendering** (ForwardBase or ForwardAdd pass types).

Function:	Description:
`float3 WorldSpaceLightDir (float4 v)`	Computes world space direction (not normalized) to light, given object space vertex position.

Function:	Description:
`float3 ObjSpaceLightDir (float4 v)`	Computes object space direction (not normalized) to light, given object space vertex position.
`float3 Shade4PointLights (...)`	Computes illumination from four point lights, with light data tightly packed into vectors. Forward **rendering** uses this to compute per-vertex lighting.

Screen-space helper functions in UnityCG.cginc

The following functions are helpers to compute coordinates used for sampling screen-space textures. They return `float4` where the final coordinate to sample texture with can be computed via perspective division (for example `xy/w`).

The functions also take care of platform differences in **render texture** coordinates.

Function:	Description:
`float4 ComputeScreenPos (float4 clipPos)`	Computes texture coordinate for doing a screenspace-mapped texture sample. Input is clip space position.
`float4 ComputeGrabScreenPos (float4 clipPos)`	Computes texture coordinate for sampling a GrabPass texure. Input is clip space position.

Vertex-lit helper functions in UnityCG.cginc

These functions are only useful when using per-vertex lit shaders (“Vertex” pass type).

Function:	Description:
`float3 ShadeVertexLights (float4 vertex, float3 normal)`	Computes illumination from four per-vertex lights and ambient, given object space position & normal.

Built-in shader variables

Leave feedback

Unity provides a handful of built-in global variables for your **shaders**: things like current object's transformation matrices, light parameters, current time and so on. You use them in shader programs like any other variable, the only difference is that you don't have to declare them - they are all declared in `UnityShaderVariables.cginc` include file that is included automatically.

Transformations

All these matrices are `float4x4` type.

Name	Value
UNITY_MATRIX_MVP	Current model * view * projection matrix.
UNITY_MATRIX_MV	Current model * view matrix.
UNITY_MATRIX_V	Current view matrix.
UNITY_MATRIX_P	Current projection matrix.
UNITY_MATRIX_VP	Current view * projection matrix.
UNITY_MATRIX_T_MV	Transpose of model * view matrix.
UNITY_MATRIX_IT_MV	Inverse transpose of model * view matrix.
unity_ObjectToWorld	Current model matrix.
unity_WorldToObject	Inverse of current world matrix.

Camera and screen

These variables will correspond to the Camera that is **rendering**. For example during shadowmap **rendering**, they will still refer to the Camera **component** values, and not the "virtual camera" that is used for the shadowmap projection.

Name	Type	Value
_WorldSpaceCameraPos	float3	World space position of the camera.
_ProjectionParams	float4	`x` is 1.0 (or –1.0 if currently rendering with a flipped projection matrix), `y` is the camera's near plane, `z` is the camera's far plane and `w` is 1/FarPlane.
_ScreenParams	float4	`x` is the width of the camera's target texture in **pixels**, `y` is the height of the camera's target texture in pixels, `z` is 1.0 + 1.0/width and `w` is 1.0 + 1.0/height.
_ZBufferParams	float4	Used to linearize Z buffer values. `x` is (1-far/near), `y` is (far/near), `z` is (x/far) and `w` is (y/far).
unity_OrthoParams	float4	`x` is orthographic camera's width, `y` is orthographic camera's height, `z` is unused and `w` is 1.0 when camera is orthographic, 0.0 when perspective.
unity_CameraProjection	float4x4	Camera's projection matrix.
unity_CameraInvProjection	float4x4	Inverse of camera's projection matrix.
unity_CameraWorldClipPlanes[6]	float4	Camera frustum plane world space equations, in this order: left, right, bottom, top, near, far.

Time

Name	Type	Value
_Time	float4	Time since level load (t/20, t, t*2, t*3), use to animate things inside the shaders.
_SinTime	float4	Sine of time: (t/8, t/4, t/2, t).
_CosTime	float4	Cosine of time: (t/8, t/4, t/2, t).
unity_DeltaTime	float4	Delta time: (dt, 1/dt, smoothDt, 1/smoothDt).

Lighting

Light parameters are passed to shaders in different ways depending on which Rendering Path is used, and which LightMode Pass Tag is used in the shader.

Forward rendering (`ForwardBase` and `ForwardAdd` pass types):

Name	Type	Value
_LightColor0 *(declared in Lighting.cginc)*	fixed4	Light color.
_WorldSpaceLightPos0	float4	Directional lights: (world space direction, 0). Other lights: (world space position, 1).
_LightMatrix0 *(declared in AutoLight.cginc)*	float4x4	World-to-light matrix. Used to sample cookie & attenuation textures.
unity_4LightPosX0, unity_4LightPosY0, unity_4LightPosZ0	float4	*(ForwardBase pass only)* world space positions of first four non-important point lights.
unity_4LightAtten0	float4	*(ForwardBase pass only)* attenuation factors of first four non-important point lights.
unity_LightColor	half4[4]	*(ForwardBase pass only)* colors of of first four non-important point lights.
unity_WorldToShadow	float4x4[4]	World-to-shadow matrices. One matrix for spot lights, up to four for directional light cascades.

Deferred shading and deferred lighting, used in the lighting pass shader (all declared in UnityDeferredLibrary.cginc):

Name	Type	Value
_LightColor	float4	Light color.
_LightMatrix0	float4x4	World-to-light matrix. Used to sample cookie & attenuation textures.
unity_WorldToShadow	float4x4[4]	World-to-shadow matrices. One matrix for spot lights, up to four for directional light cascades.

Spherical harmonics coefficients (used by ambient and light probes) are set up for `ForwardBase`, `PrePassFinal` and `Deferred` pass types. They contain 3rd order SH to be evaluated by world space normal (see `ShadeSH9` from UnityCG.cginc). The variables are all half4 type, `unity_SHAr` and similar names.

Vertex-lit rendering (`Vertex` pass type):

Up to 8 lights are set up for a `Vertex` pass type; always sorted starting from the brightest one. So if you want to render objects affected by two lights at once, you can just take first two entries in the arrays. If there are less lights affecting the object than 8, the rest will have their color set to black.

Name	Type	Value
unity_LightColor	half4[8]	Light colors.
unity_LightPosition	float4[8]	View-space light positions. (-direction,0) for directional lights; (position,1) for point/spot lights.
unity_LightAtten	half4[8]	Light attenuation factors. *x* is cos(spotAngle/2) or –1 for non-spot lights; *y* is 1/cos(spotAngle/4) or 1 for non-spot lights; *z* is quadratic attenuation; *w* is squared light range.
unity_SpotDirection	float4[8]	View-space spot light positions; (0,0,1,0) for non-spot lights.

Fog and Ambient

Name	Type	Value
unity_AmbientSky	fixed4	Sky ambient lighting color in gradient ambient lighting case.
unity_AmbientEquator	fixed4	Equator ambient lighting color in gradient ambient lighting case.
unity_AmbientGround	fixed4	Ground ambient lighting color in gradient ambient lighting case.
UNITY_LIGHTMODEL_AMBIENT	fixed4	Ambient lighting color (sky color in gradient ambient case). Legacy variable.
unity_FogColor	fixed4	Fog color.

unity_FogParams	float4	Parameters for fog calculation: (density / sqrt(ln(2)), density / ln(2), –1/(end-start), end/(end-start)). *x* is useful for Exp2 fog mode, *y* for Exp mode, *z* and *w* for Linear mode.

Various

Name	Type	Value
unity_LODFade	float4	Level-of-detail fade when using LODGroup. *x* is fade (0..1), *y* is fade quantized to 16 levels, *z* and *w* unused.
_TextureSampleAdd	float4	Set automatically by Unity **for UI only** based on whether the texture being used is in Alpha8 format (the value is set to (1,1,1,0)) or not (the value is set to (0,0,0,0)).

Making multiple shader program variants

Leave feedback

Often it is convenient to keep most of a piece of **shader** code fixed but also allow slightly different **shader** "variants" to be produced. This is commonly called "mega shaders" or "uber shaders", and is achieved by compiling the **shader** code multiple times with different preprocessor directives for each case.

In Unity this can be achieved by adding a `#pragma multi_compile` or `#pragma shader_feature` directive to a shader snippet. This works in surface shaders too.

At runtime, the appropriate shader variant is picked up from the Material keywords (Material.EnableKeyword and DisableKeyword) or global shader keywords (Shader.EnableKeyword and DisableKeyword).

How multi_compile works

A directive like:

```
#pragma multi_compile FANCY_STUFF_OFF FANCY_STUFF_ON
```

Will produce two shader variants, one with `FANCY_STUFF_OFF` defined, and another with `FANCY_STUFF_ON`. At runtime, one of them will be activated based on the Material or global shader keywords. If neither of these two keywords are enabled then the first one ("off") will be used.

There can be more than two keywords on a multi_compile line, for example this will produce four shader variants:

```
#pragma multi_compile SIMPLE_SHADING BETTER_SHADING GOOD_SHADING BEST_SHADING
```

When any of the names are all underscores, then a shader variant will be produced, with no preprocessor macro defined. This is commonly used for shaders features, to avoid using up two keywords (see notes on keywork limit below). For example, the directive below will produce two shader variants; first one with nothing defined, and second one with `FOO_ON` defined:

```
#pragma multi_compile __ FOO_ON
```

Difference between shader_feature and multi_compile

`#pragma shader_feature` is very similar to `#pragma multi_compile`, the only difference is that unused variants of shader_feature shaders will not be included into game build. So shader_feature makes most sense for keywords that will be set on the materials, while multi_compile for keywords that will be set from code globally.

Additionally, it has a shorthand notation with just one keyword:

```
#pragma shader_feature FANCY_STUFF
```

Which is just a shortcut for `#pragma shader_feature _ FANCY_STUFF`, i.e. it expands into two shader variants (first one without the define; second one with it).

Combining several multi_compile lines

Several multi_compile lines can be provided, and the resulting shader will be compiled for all possible combinations of the lines:

```
#pragma multi_compile A B C
#pragma multi_compile D E
```

This would produce three variants for first line, and two for the second line, or in total six shader variants (A+D, B+D, C+D, A+E, B+E, C+E).

It's easiest to think of each multi_compile line as controlling a single shader "feature". Keep in mind that the total number of shader variants grows really fast this way. For example, ten multi_compile "features" with two options each produces 1024 shader variants in total!

Keyword limit

When using Shader variants, remember that there is a limit of 256 keywords in Unity, and around 60 of them are used internally (therefore lowering the available limit). Also, the keywords are enabled globally throughout a particular Unity project, so be careful not to exceed the limit when multiple keywords are defined in several different Shaders.

Built-in multi_compile shortcuts

There are several "shortcut" notations for compiling multiple shader variants; they are mostly to deal with different light, shadow and lightmap types in Unity. See rendering pipeline for details.

`multi_compile_fwdbase` compiles all variants needed by `ForwardBase` (forward rendering base) pass type. The variants deal with different lightmap types and main directional light having shadows on or off.

`multi_compile_fwdadd` compiles variants for `ForwardAdd` (forward rendering additive) pass type. This compiles variants to handle directional, spot or point light types, and their variants with cookie textures.
`multi_compile_fwdadd_fullshadows` - same as above, but also includes ability for the lights to have realtime shadows.
`multi_compile_fog` expands to several variants to handle different fog types (off/linear/exp/exp2).
Most of the built-in shortcuts result in many shader variants. It is possible to skip compiling some of them if you know they are not needed, by using `#pragma skip_variants`. For example:

```
#pragma multi_compile_fwdadd
// will make all variants containing
// "POINT" or "POINT_COOKIE" be skipped
#pragma skip_variants POINT POINT_COOKIE
```

Shader Hardware Variants

One common reason for using shader variants is to create fallbacks or simplified shaders that can run efficiently on both high and low end hardware within a single target platform - such as OpenGL ES. To provide a specially optimised set of variants for different levels of hardware capability, you can use shader hardware variants.

To enable the generation of shader hardware variants, add `#pragma hardware_tier_variants renderer`, where `renderer` is one of the available renderering platforms for shader program pragmas. With this `#pragma` 3 shader variants will be generated for each shader, regardless of any other keywords. Each variant will have one of the following defined:

```
UNITY_HARDWARE_TIER1
UNITY_HARDWARE_TIER2
UNITY_HARDWARE_TIER3
```

You can use these to write conditional fallbacks or extra features for lower or higher end. In the editor you can test any of the tiers by using the Graphics Emulation menu, which allows you to change between each of the tiers.

To help keep the impact of these variants as small as possible, only one set of shaders is ever loaded in the player. In addition, any shaders which end up identical - for example if you only write a specialised version for TIER1 and all others are the same - will not take up any extra space on disk.

At load time Unity will examine the GPU that it is using and auto-detect a tier value; it will default to the highest tier if the GPU is not auto-detected. You can override this tier value by setting `Shader.globalShaderHardwareTier`, but this must be done before any shaders you want to vary are loaded.

Once the shaders are loaded they will have selected their set of variants and this value will have no effect. A good place to set this would be in a pre-load scene before you load your main scene.

Note that these shader hardware tiers are not related to the quality settings of the player, they are purely detected from the relative capability of the GPU the player is running on.

Platform Shader Settings

Apart from tweaking your shader code for different hardware tiers, you might want to tweak unity internal defines (e.g. you might want to force cascaded shadowmaps on mobiles). You can find details on this in the UnityEditor.Rendering.PlatformShaderSettings documentation, which provides a list of currently supported features for overriding per-tier. Use UnityEditor.Rendering.EditorGraphicsSettings.SetShaderSettingsForPlatform to tweak Platform Shader Settings per-platform per-tier.

Please note that if `PlatformShaderSettings` set to different tiers are not identical, then tier variants will be generated for the shader even if `#pragma hardware_tier_variants` is missing.

See Also

Optimizing Shader Load Time.

GLSL Shader programs

Leave feedback

In addition to using Cg/HSL shader programs, OpenGL Shading Language (GLSL) **Shaders** can be written directly.

However, use of raw GLSL is only recommended for testing, or when you know you are only targeting Mac OS X, OpenGL ES mobile devices, or Linux. In all normal cases, Unity will cross-compile Cg/HLSL into optimized GLSL when needed.

GLSL snippets

GLSL program snippets are written between `GLSLPROGRAM` and `ENDGLSL` keywords.

In GLSL, all shader function entry points have to be called `main()`. When Unity loads the GLSL shader, it loads the source once for the vertex program, with the `VERTEX` preprocessor define, and once more for the fragment program, with the `FRAGMENT` preprocessor define. So the way to separate vertex and fragment program parts in GLSL snippet is to surround them with `#ifdef VERTEX .. #endif` and `#ifdef FRAGMENT .. #endif`. Each GLSL snippet must contain both a vertex program and a fragment program.

Standard include files match those provided for Cg/HLSL shaders; they just have a `.glslinc` extension:

```
UnityCG.glslinc
```

Vertex shader inputs come from predefined GLSL variables (`gl_Vertex`, `gl_MultiTexCoord0`, ...) or are user defined attributes. Usually only the tangent vector needs a user defined attribute:

```
attribute vec4 Tangent;
```

Data from vertex to fragment programs is passed through *varying* variables, for example:

```
varying vec3 lightDir; // vertex shader computes this, fragment shader uses this
```

External OES textures

Unity does some preprocessing during Shader compilation; for example, `texture2D/texture2DProj` functions may be replaced to `texture/textureProj`, based on graphics API (GlES3, GLCore). Some extensions don't support new convention, most notably `GL_OES_EGL_image_external`.

If you want to sample external textures in GLSL shaders, use `textureExternal/textureProjExternal` calls instead of `texture2D/texture2DProj`.

Example:

```
gl_FragData[0] = textureExternal(_MainTex, uv);
```

Shading Language used in Unity

Leave feedback

In Unity, shader programs are written in a variant of HLSL language (also called Cg but for most practical uses the two are the same).

Currently, for maximum portability between different platforms, writing in DX9-style HLSL (e.g. use DX9 style `sampler2D` and `tex2D` for texture sampling instead of DX10 style `Texture2D`, `SamplerState` and `tex.Sample`).

Shader Compilers

Internally, different **shader** compilers are used for shader program compilation:

Windows & Microsoft platforms (DX11, DX12 and Xbox One) all use Microsoft's HLSL compiler (currently d3dcompiler_47).
OpenGL Core, OpenGL ES 3, OpenGL ES 2.0 and Metal use Microsoft's HLSL followed by bytecode translation into GLSL or Metal, using HLSLcc.
OpenGL ES 2.0 can use source level translation via hlsl2glslfork and glsl optimizer. This is enabled by adding `#pragma prefer_hlsl2glsl gles`
Other console platforms use their respective compilers (e.g. PSSL on PS4).
Surface Shaders use Cg 2.2 and MojoShader for code generation analysis step.
In case you really need to identify which compiler is being used (to use HLSL syntax only supported by one compiler, or to work around a compiler bug), predefined shader macros can be used. For example, `UNITY_COMPILER_HLSL` is set when compiling with HLSL compiler (for D3D or GLCore/GLES3/GLES platforms); and `UNITY_COMPILER_HLSL2GLSL` when compiling via hlsl2glsl.

See Also

Shader Programs.
Shader Preprocessor Macros.
Platform Specific Rendering Differences.

Shader Compilation Target Levels

Leave feedback

When writing either Surface Shaders or regular Shader Programs, the HLSL source can be compiled into different "**shader** models". To allow the use of more modern GPI functionality, you must use higher shader compilation targets.

Note: Using higher shader compilation targets may prevent the shader from working on older GPUs or platforms.

Indicate the compilation target by using the `#pragma target` *name* directive. For example:

```
#pragma target 3.5
```

Default compilation target

By default, Unity compiles shaders into almost the lowest supported target ("2.5"); in between DirectX shader models 2.0 and 3.0. Some other compilation directives make the shader automatically be compiled into a higher target:

Using a geometry shader (`#pragma geometry`) sets the compilation target to `4.0`.
Using tessellation shaders (`#pragma hull` or `#pragma domain`) sets the compilation target to `4.6`.
Any shader not explicitly setting a function entry point through `#pragma` for geometry, hull or domain shaders will downgrade internal shader capability requirements. This allows non-DX11 targets with broader run-time and feature differences to be more compatible with the existing shader content.

For example, Unity supports tessellation shaders on Metal graphics, but Metal doesn't support geometry shaders. Using `#pragma target 5.0` is still valid, as long as you don't use geometry shaders.

Supported #pragma target names

Here is the list of shader models supported, with roughly increasing set of capabilities (and in some cases higher platform/GPU requirements):

#pragma target 2.0

Works on all platforms supported by Unity. DX9 shader model 2.0.
Limited amount of arithmetic & texture instructions; 8 interpolators; no vertex texture sampling; no derivatives in **fragment shaders**; no explicit **LOD** texture sampling.

#pragma target 2.5 (default)

Almost the same as 3.0 target (see below), except still only has 8 interpolators, and does not have explicit LOD texture sampling.
Compiles into DX11 feature level 9.3 on Windows Phone.

#pragma target 3.0

DX9 shader model 3.0: derivative instructions, texture LOD sampling, 10 interpolators, more math/texture instructions allowed.

Not supported on DX11 feature level 9.x GPUs (e.g. most Windows Phone devices).
Might not be fully supported by some OpenGL ES 2.0 devices, depending on driver extensions present and features used.

#pragma target 3.5 (or es3.0)

OpenGL ES 3.0 capabilities (DX10 SM4.0 on D3D platforms, just without geometry shaders).
Not supported on DX11 9.x (WinPhone), OpenGL ES 2.0.
Supported on DX11+, OpenGL 3.2+, OpenGL ES 3+, Metal, Vulkan, **PS4**/XB1 consoles.
Native integer operations in shaders, texture arrays, **etc**.

#pragma target 4.0

DX11 shader model 4.0.
Not supported on DX11 9.x (WinPhone), OpenGL ES 2.0/3.0/3.1, Metal.
Supported on DX11+, OpenGL 3.2+, OpenGL ES 3.1+AEP, Vulkan, PS4/XB1 consoles.
Has geometry shaders and everything that `es3.0` target has.

#pragma target 4.5 (or es3.1)

OpenGL ES 3.1 capabilities (DX11 SM5.0 on D3D platforms, just without tessellation shaders).
Not supported on DX11 before SM5.0, OpenGL before 4.3 (i.e. Mac), OpenGL ES 2.0/3.0.
Supported on DX11+ SM5.0, OpenGL 4.3+, OpenGL ES 3.1, Metal, Vulkan, PS4/XB1 consoles.
Has compute shaders, random access texture writes, atomics etc. No geometry or tessellation shaders.

#pragma target 4.6 (or gl4.1)

OpenGL 4.1 capabilities (DX11 SM5.0 on D3D platforms, just without compute shaders). This is basically the highest OpenGL level supported by Macs.
Not supported on DX11 before SM5.0, OpenGL before 4.1, OpenGL ES 2.0/3.0/3.1, Metal.
Supported on DX11+ SM5.0, OpenGL 4.1+, OpenGL ES 3.1+AEP, Vulkan, Metal (without geometry), PS4/XB1 consoles.

#pragma target 5.0

DX11 shader model 5.0.
Not supported on DX11 before SM5.0, OpenGL before 4.3 (i.e. Mac), OpenGL ES 2.0/3.0/3.1, Metal.
Supported on DX11+ SM5.0, OpenGL 4.3+, OpenGL ES 3.1+AEP, Vulkan, Metal (without geometry), PS4/XB1 consoles.

Note that all OpenGL-like platforms (including mobile) are treated as “capable of shader model 3.0”. WP8/WinRT platforms (DX11 feature level 9.x) are treated as only capable of shader model 2.5.

See Also

Writing Shader Programs.
Surface Shaders.

2018–03–20 Page amended with editorial review

Shader #pragma directives added in Unity 2018.1

Tessellation for Metal added in 2018.1

Shader data types and precision

Leave feedback

The **standard Shader** language in Unity is HLSL, and general HLSL data types are supported. However, Unity has some additions to the HLSL types, particularly for better support on mobile platforms.

Basic data types

The majority of calculations in **shaders** are carried out on floating-point numbers (which would be `float` in regular programming languages like C#). Several variants of floating point types are present: `float`, `half` and `fixed` (as well as vector/matrix variants of them, such as `half3` and `float4x4`). These types differ in precision (and, consequently, performance or power usage):

High precision: `float`

Highest precision floating point value; generally 32 bits (just like `float` from regular programming languages).

Full `float` precision is generally used for world space positions, texture coordinates, or scalar computations involving complex functions such as trigonometry or power/exponentiation.

Medium precision: `half`

Medium precision floating point value; generally 16 bits (range of –60000 to +60000, with about 3 decimal digits of precision).

Half precision is useful for short vectors, directions, object space positions, high dynamic range colors.

Low precision: `fixed`

Lowest precision fixed point value. Generally 11 bits, with a range of –2.0 to +2.0 and 1/256th precision.

Fixed precision is useful for regular colors (as typically stored in regular textures) and performing simple operations on them.

Integer data types

Integers (`int` data type) are often used as loop counters or array indices. For this purpose, they generally work fine across various platforms.

Depending on the platform, integer types might not be supported by the GPU. For example, Direct3D 9 and OpenGL ES 2.0 GPUs only operate on floating point data, and simple-looking integer expressions (involving bit or logical operations) might be emulated using fairly complicated floating point math instructions.

Direct3D 11, OpenGL ES 3, Metal and other modern platforms have proper support for integer data types, so using bit shifts and bit masking works as expected.

Composite vector/matrix types

HLSL has built-in vector and matrix types that are created from the basic types. For example, `float3` is a 3D vector with .x, .y, .z **components**, and `half4` is a medium precision 4D vector with .x, .y, .z, .w components. Alternatively, vectors can be indexed using .r, .g, .b, .a components, which is useful when working on colors.

Matrix types are built in a similar way; for example `float4x4` is a 4x4 transformation matrix. Note that some platforms only support square matrices, most notably OpenGL ES 2.0.

Texture/Sampler types

Typically you declare textures in your HLSL code as follows:

```
sampler2D _MainTex;
samplerCUBE _Cubemap;
```

For mobile platforms, these translate into "low precision samplers", i.e. the textures are expected to have low precision data in them. If you know your texture contains **HDR** colors, you might want to use half precision sampler:

```
sampler2D_half _MainTex;
samplerCUBE_half _Cubemap;
```

Or if your texture contains full float precision data (e.g. depth texture), use a full precision sampler:

```
sampler2D_float _MainTex;
samplerCUBE_float _Cubemap;
```

Precision, Hardware Support and Performance

One complication of `float`/`half`/`fixed` data type usage is that PC GPUs are **always** high precision. That is, for all the PC (Windows/Mac/Linux) GPUs, it does not matter whether you write `float`, `half` or `fixed` data types in your shaders. They always compute everything in full 32-bit floating point precision.

The `half` and `fixed` types only become relevant when targeting mobile GPUs, where these types primarily exist for power (and sometimes performance) constraints. Keep in mind that you need to test your shaders on mobile to see whether or not you are running into precision/numerical issues.

Even on mobile GPUs, the different precision support varies between GPU families. Here's an overview of how each mobile GPU family treats each floating point type (indicated by the number of bits used for it):

GPU Family	float	half	fixed
PowerVR Series 6/7	32	16	
PowerVR SGX 5xx	32	16	11
Qualcomm Adreno 4xx/3xx	32	16	
Qualcomm Adreno 2xx	32 vertex 24 fragment		
ARM Mali T6xx/7xx	32	16	
ARM Mali 400/450	32 vertex 16 fragment		
NVIDIA X1	32	16	

GPU Family	float	half	fixed
NVIDIA K1	32		
NVIDIA Tegra 3/4	32	16	

Most modern mobile GPUs actually only support either 32-bit numbers (used for `float` type) or 16-bit numbers (used for both `half` and `fixed` types). Some older GPUs have different precisions for **vertex shader** and **fragment shader** computations.

Using lower precision can often be faster, either due to improved GPU register allocation, or due to special "fast path" execution units for certain lower-precision math operations. Even when there's no raw performance advantage, using lower precision often uses less power on the GPU, leading to better battery life.

A general rule of thumb is to start with half precision for everything except positions and texture coordinates. Only increase precision if half precision is not enough for some parts of the computation.

Support for infinities, NaNs and other special floating point values

Support for special floating point values can be different depending on which (mostly mobile) GPU family you're running.

All PC GPUs that support Direct3D 10 support very well-specified IEEE 754 floating point standard. This means that float numbers behave exactly like they do in regular programming languages on the CPU.

Mobile GPUs can have slightly different levels of support. On some, dividing zero by zero might result in a NaN ("not a number"); on others it might result in infinity, zero or any other unspecified value. Make sure to test your shaders on the target device to check they are supported.

External GPU Documentation

GPU vendors have in-depth guides about the performance and capabilities of their GPUs. See these for details:

ARM Mali Guide for Unity Developers
Qualcomm Adreno OpenGL ES Developer Guide
PowerVR Architecture Guides

See Also

Platform Specific Rendering Differences
Shader Performance Tips
Shading Language

Using sampler states

Leave feedback

Coupled textures and samplers

Most of the time when sampling textures in **shaders**, the texture sampling state should come form texture settings – essentially, textures and samplers are coupled together. This is default behavior when using DX9-style shader syntax:

```
sampler2D _MainTex;
// ...
half4 color = tex2D(_MainTex, uv);
```

Using sampler2D, sampler3D, samplerCUBE HLSL keywords declares both texture and sampler.

Most of the time this is what you want, and is the only supported option on older graphics APIs (OpenGL ES).

Separate textures and samplers

Many graphics APIs and GPUs allow using fewer samplers than textures, and coupled texture+sampler syntax might not allow more complex shaders to be written. For example Direct3D 11 allows using up to 128 textures in a single shader, but only up to 16 samplers.

Unity allows declaring textures and samplers using DX11-style HLSL syntax, with a special naming convention to match them up: samplers that have names in the form of “sampler”+TextureName will take sampling states from that texture.

The shader snippet from section above could be rewritten in DX11-style HLSL syntax, and it would do the same thing:

```
Texture2D _MainTex;
SamplerState sampler_MainTex; // "sampler" + “_MainTex”
// ...
half4 color = _MainTex.Sample(sampler_MainTex, uv);
```

However, this way a shader could be written to “reuse” samplers from other textures, while sampling more than one texture. In the example below, three textures are sampled, but only one sampler is used for all of them:

```
Texture2D _MainTex;
Texture2D _SecondTex;
Texture2D _ThirdTex;
```

```
SamplerState sampler_MainTex; // "sampler" + "_MainTex"
// ...
half4 color = _MainTex.Sample(sampler_MainTex, uv);
color += _SecondTex.Sample(sampler_MainTex, uv);
color += _ThirdTex.Sample(sampler_MainTex, uv);
```

Note however that DX11-style HLSL syntax does not work on some older platforms (e.g. OpenGL ES 2.0), see shading language for details. You might want to specify `#pragma target 3.5` (see shader compilation targets to skip older platforms from using the shader.

Unity provides several shader macros to help with declaring and sampling textures using this "separate samplers" approach, see built-in macros. The example above could be rewritten this way, using said macros:

```
UNITY_DECLARE_TEX2D(_MainTex);
UNITY_DECLARE_TEX2D_NOSAMPLER(_SecondTex);
UNITY_DECLARE_TEX2D_NOSAMPLER(_ThirdTex);
// ...
half4 color = UNITY_SAMPLE_TEX2D(_MainTex, uv);
color += UNITY_SAMPLE_TEX2D_SAMPLER(_SecondTex, _MainTex, uv);
color += UNITY_SAMPLE_TEX2D_SAMPLER(_ThirdTex, _MainTex, uv);
```

The above would compile on all platforms supported by Unity, but would fallback to using three samplers on older platforms like DX9.

Inline sampler states

In addition to recognizing HLSL SamplerState objects named as "sampler"+TextureName, Unity also recognizes some other patterns in sampler names. This is useful for declaring simple hardcoded sampling states directly in the shaders. An example:

```
Texture2D _MainTex;
SamplerState my_point_clamp_sampler;
// ...
half4 color = _MainTex.Sample(my_point_clamp_sampler, uv);
```

The name "my_point_clamp_sampler" will be recognized as a sampler that should use Point (nearest) texture filtering, and Clamp texture wrapping mode.

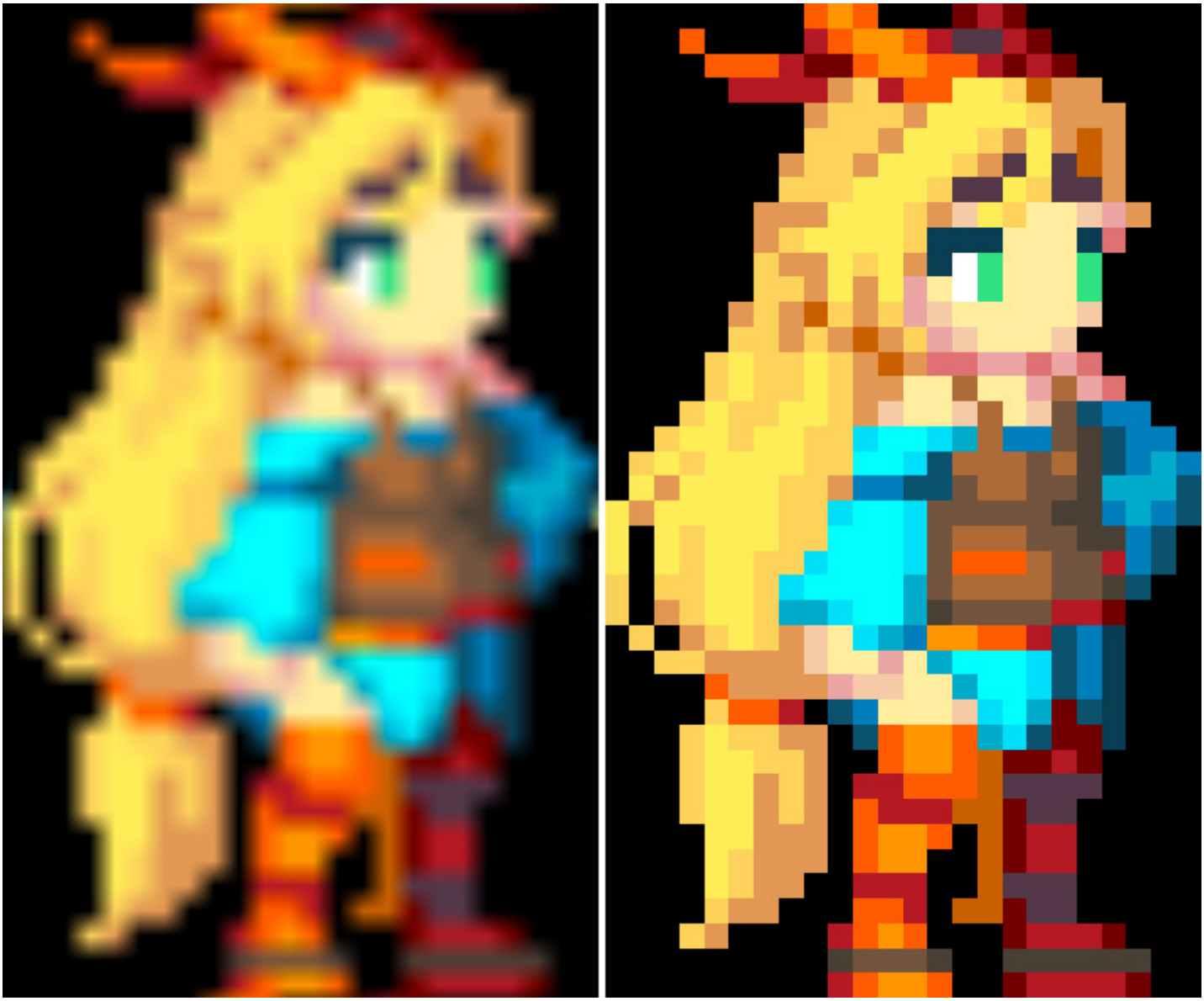
Sampler names recognized as “inline” sampler states (all case insensitive):

“Point”, “Linear” or “Trilinear” (required) set up texture filtering mode.

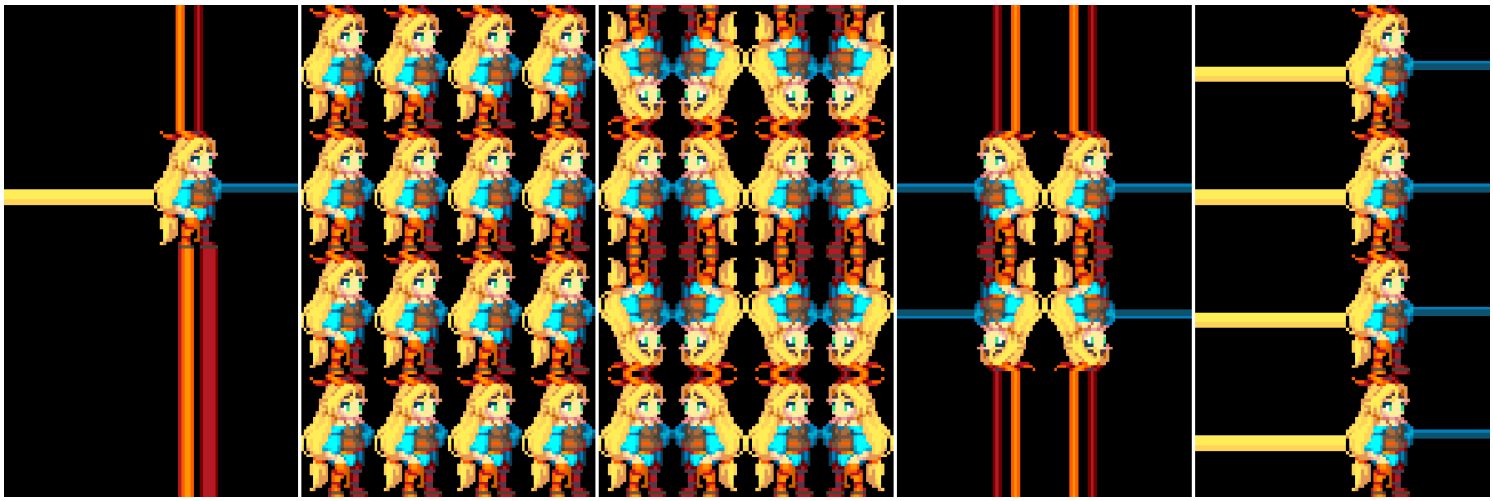
“Clamp”, “Repeat”, “Mirror” or “MirrorOnce” (required) set up texture wrap mode.

Wrap modes can be specified per-axis (UVW), e.g. “ClampU_RepeatV”.
“Compare” (optional) set up sampler for depth comparison; use with HLSL SamplerComparisonState type and SampleCmp / SampleCmpLevelZero functions.

Here’s an example of sampling texture with `sampler_linear_repeat` and `sampler_point_repeat` SamplerStates respectively, illustrating how the name controls filtering mode:



Here’s an example with `SmpClampPoint`, `SmpRepeatPoint`, `SmpMirrorPoint`, `SmpMirrorOncePoint`, `Smp_ClampU_RepeatV_Point` SamplerStates respectively, illustrating how the name controls wrapping mode. In the last example, different wrap modes are set for horizontal (U) and vertical (V) axes. In all cases texture coordinates go from –2.0 to +2.0.



Just like separate texture + sampler syntax, inline sampler states are not supported on some platforms. Currently they are implemented on Direct3D 11/12, **PS4**, XboxOne and Metal.

Note that “MirrorOnce” texture wrapping mode is not supported on most mobile GPUs/APIs, and will fallback to Mirror mode when support is not present.

2017–06–01 Page published with no editorial review

New feature in 2017.1

ShaderLab Syntax

Leave feedback

All Shaders files in Unity are written in a declarative language called "**ShaderLab**". In the file, a nested-braces syntax declares various things that describe the shader – for example which shader properties should be shown in material **inspector**; what kind of hardware fallbacks to do; what kind of blending modes to use **etc**.; and actual "shader code" is written in `CGPROGRAM` snippets inside the same shader file (see surface shaders and vertex and fragment shaders).

This page and the child pages describes the nested-braces "ShaderLab" syntax. The `CGPROGRAM` snippets are written in regular HLSL/Cg shading language, see their documentation pages.

Shader is the root command of a shader file. Each file must define one (and only one) Shader. It specifies how any objects whose material uses this shader are rendered.

Syntax

```
Shader "name" { [Properties] __Subshaders__ [Fallback] [CustomEditor] }
```

Defines a shader. It will appear in the material inspector listed under **name**. Shaders optionally can define a list of **properties** that show up in material inspector. After this comes a list of SubShaders, and optionally a fallback and/or a custom editor declaration.

Details

Properties

Shaders can have a list of properties. Any properties declared in a shader are shown in the material inspector inside Unity. Typical properties are the object color, textures, or just arbitrary values to be used by the shader.

SubShaders & Fallback

Each shader is comprised of a list of sub-shaders. You must have at least one. When loading a shader, Unity will go through the list of subshaders, and pick the first one that is supported by the end user's machine. If no subshaders are supported, Unity will try to use fallback shader.

Different graphic cards have different capabilities. This raises an eternal issue for game developers; you want your game to look great on the latest hardware, but don't want it to be available only to those 3% of the population. This is where subshaders come in. Create one subshader that has all the fancy graphic effects you can dream of, then add more subshaders for older cards. These subshaders may implement the effect you want in a slower way, or they may choose not to implement some details.

Shader "level of detail" (LOD) and "shader replacement" are two techniques that also build upon subshaders, see Shader LOD and Shader Replacemement for details.

Examples

Here is one of the simplest shaders possible:

```
// colored vertex lighting
Shader "Simple colored lighting"
{
```

```
    // a single color property
    Properties {
        _Color ("Main Color", Color) = (1,.5,.5,1)
    }
    // define one subshader
    SubShader
    {
        // a single pass in our subshader
        Pass
        {
            // use fixed function per-vertex lighting
            Material
            {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

This shader defines a color property **_Color** (that shows up in material inspector as *Main Color*) with a default value of **(1,0.5,0.5,1)**. Then a single subshader is defined. The subshader consists of one Pass that turns on fixed-function vertex lighting and sets up basic material for it.

See more complex examples at Surface Shader Examples or Vertex and Fragment Shader Examples.

See Also

Properties Syntax.
SubShader Syntax.
Pass Syntax.
Fallback Syntax.
CustomEditor Syntax.

ShaderLab: Properties

Leave feedback

Shaders can define a list of parameters to be set by artists in Unity's material inspector. The Properties block in the shader file defines them.

Syntax

Properties

```
Properties { Property [Property ...] }
```

Defines the property block. Inside braces multiple properties are defined as follows.

Numbers and Sliders

```
name ("display name", Range (min, max)) = number
name ("display name", Float) = number
name ("display name", Int) = number
```

These all defines a number (scalar) property with a default value. The `Range` form makes it be displayed as a slider between *min* and *max* ranges.

Colors and Vectors

```
name ("display name", Color) = (number,number,number,number)
name ("display name", Vector) = (number,number,number,number)
```

Defines a color property with default value of given RGBA **components**, or a 4D vector property with a default value. Color properties have a color picker shown for them, and are adjusted as needed depending on the color space (see Properties in Shader Programs). Vector properties are displayed as four number fields.

Textures

```
name ("display name", 2D) = "defaulttexture" {}
name ("display name", Cube) = "defaulttexture" {}
name ("display name", 3D) = "defaulttexture" {}
```

Defines a 2D texture, cubemap or 3D (volume) property respectively.

Details

Each property inside the **shader** is referenced by **name** (in Unity, it's common to start shader property names with underscore). The property will show up in material **inspector** as **display name**. For each property a default value is given after equals sign:

For *Range* and *Float* properties it's just a single number, for example "13.37".
For *Color* and *Vector* properties it's four numbers in parentheses, for example "(1,0.5,0.2,1)".
For 2D Textures, the default value is either an empty string, or one of the built-in default Textures: "white" (RGBA: 1,1,1,1), "black" (RGBA: 0,0,0,0), "gray" (RGBA: 0.5,0.5,0.5,0.5), "bump" (RGBA: 0.5,0.5,1,0.5) or "red" (RGBA: 1,0,0,0).
For non–2D Textures (Cube, 3D, 2DArray) the default value is an empty string. When a Material does not have a Cubemap/3D/Array Texture assigned, a gray one (RGBA: 0.5,0.5,0.5,0.5) is used.
Later on in the shader's fixed function parts, property values can be accessed using property name in square brackets: **[name]**. For example, you could make blending mode be driven by a material property by declaring two integer properties (say "*SrcBlend" and "*DstBlend"), and later on make Blend Command use them: `Blend [_SrcBlend] [_DstBlend]`.

Shader parameters that are in the `Properties` block are serialized as Material data. Shader programs can actually have more parameters (like matrices, vectors and floats) that are set on the material from code at runtime, but if they are not part of the Properties block then their values will not be saved. This is mostly useful for values that are completely script code-driven (using Material.SetFloat and similar functions).

Property attributes and drawers

In front of any property, optional attributes in square brackets can be specified. These are either attributes recognized by Unity, or they can indicate your own MaterialPropertyDrawer classes to control how they should be rendered in the material inspector. Attributes recognized by Unity:

`[HideInInspector]` - does not show the property value in the material inspector.
`[NoScaleOffset]` - material inspector will not show texture tiling/offset fields for texture properties with this attribute.
`[Normal]` - indicates that a texture property expects a normal-map.
`[HDR]` - indicates that a texture property expects a high-dynamic range (HDR) texture.
`[Gamma]` - indicates that a float/vector property is specified as sRGB value in the **UI** (just like colors are), and possibly needs conversion according to color space used. See Properties in Shader Programs.
`[PerRendererData]` - indicates that a texture property will be coming from per-renderer data in the form of a MaterialPropertyBlock. Material inspector changes the texture slot UI for these properties.

Example

```
// properties for a water shader
Properties
{
    _WaveScale ("Wave scale", Range (0.02,0.15)) = 0.07 // sliders
    _ReflDistort ("Reflection distort", Range (0,1.5)) = 0.5
    _RefrDistort ("Refraction distort", Range (0,1.5)) = 0.4
    _RefrColor ("Refraction color", Color) = (.34, .85, .92, 1) // color
    _ReflectionTex ("Environment Reflection", 2D) = "" {} // textures
```

```
    _RefractionTex ("Environment Refraction", 2D) = "" {}
    _Fresnel ("Fresnel (A) ", 2D) = "" {}
    _BumpMap ("Bumpmap (RGB) ", 2D) = "" {}
}
```

Texture property options (removed in 5.0)

Before Unity 5, texture properties could have options inside the curly brace block, e.g. `TexGen CubeReflect`. These were controlling fixed function texture coordinate generation. This functionality was removed in 5.0; if you need texgen you should write a vertex shader instead. See Implementing Fixed Function TexGen page page for examples.

See Also

Accessing Shader Properties in HLSL/Cg.
Material Property Drawers.

ShaderLab: SubShader

Leave feedback

Each **shader** in Unity consists of a list of **subshaders**. When Unity has to display a **mesh**, it will find the **shader** to use, and pick the first subshader that runs on the user's graphics card.

Syntax

```
Subshader { [Tags] [CommonState] Passdef [Passdef ...] }
```

Defines the subshader as optional tags, common state and a list of pass definitions.

Details

A subshader defines a list of rendering passes and optionally setup any state that is common to all passes. Additionally, subshader specific Tags can be set up.

When Unity chooses which subshader to render with, it renders an object once for each Pass defined (and possibly more due to light interactions). As each render of the object is an expensive operation, you want to define the shader in minimum amount of passes possible. Of course, sometimes on some graphics hardware the needed effect can't be done in a single pass; then you have no choice but to use multiple passes.

Each pass definition can be a regular Pass, a Use Pass or a Grab Pass.

Any statements that are allowed in a Pass definition can also appear in Subshader block. This will make all passes use this "shared" state.

Example

```
// ...
SubShader {
    Pass {
        Lighting Off
        SetTexture [_MainTex] {}
    }
}
// ...
```

This subshader defines a single Pass that turns off any lighting and just displays a mesh with texture named **_MainTex**.

ShaderLab: Pass

Leave feedback

The Pass block causes the geometry of a **GameObject** to be rendered once.

Syntax

```
Pass { [Name and Tags] [RenderSetup] }
```

The basic Pass command contains a list of render state setup commands.

Name and tags

A Pass can define its Name and arbitrary number of Tags. These are name/value strings that communicate the Pass's intent to the **rendering** engine.

Render state set-up

A Pass sets up various states of the graphics hardware, such as whether alpha blending should be turned on or whether depth testing should be used.

The commands are as follows:

Cull

```
Cull Back | Front | Off
```

Set polygon culling mode.

ZTest

```
ZTest (Less | Greater | LEqual | GEqual | Equal | NotEqual | Always)
```

Set depth buffer testing mode.

ZWrite

```
ZWrite On | Off
```

Set depth buffer writing mode.

Offset

```
Offset OffsetFactor, OffsetUnits
```

Set Z buffer depth offset. See documentation on Cull and Depth page for more details.

See documentation on Cull and Depth for more details on Cull, ZTest, ZWrite and Offset.

Blend

```
Blend sourceBlendMode destBlendMode
Blend sourceBlendMode destBlendMode, alphaSourceBlendMode alphaDestBlendMode
BlendOp colorOp
BlendOp colorOp, alphaOp
AlphaToMask On | Off
```

Sets alpha blending, alpha operation, and alpha-to-coverage modes. See documentation on Blending for more details.

ColorMask

```
ColorMask RGB | A | 0 | any combination of R, G, B, A
```

Set color channel writing mask. Writing *ColorMask 0* turns off rendering to all color channels. Default mode is writing to all channels (RGBA), but for some special effects you might want to leave certain channels unmodified, or disable color writes completely.

When using multiple render target (MRT) rendering, it is possible to set up different color masks for each render target, by adding index (0–7) at the end. For example, `ColorMask RGB 3` would make render target #3 write only to RGB channels.

Legacy fixed-function Shader commands

A number of commands are used for writing legacy "fixed-function style" **Shaders**. This is considered deprecated functionality, as writing Surface Shaders or Shader programs allows much more flexibility. However, for very

simple Shaders, writing them in fixed-function style can sometimes be easier, so the commands are provided here. Note that all of the following commands are are ignored if you are not using fixed-function Shaders.

Fixed-function Lighting and Material

```
Lighting On | Off
Material { Material Block }
SeparateSpecular On | Off
Color Color-value
ColorMaterial AmbientAndDiffuse | Emission
```

All of these control fixed-function per-vertex Lighting: they turn it on, set up Material colors, turn on specular highlights, provide default color (if vertex Lighting is off), and controls how the **mesh** vertex colors affect Lighting. See documentation on Materials for more details.

Fixed-function Fog

```
Fog { Fog Block }
```

Set fixed-function Fog parameters. See documentation onFogging for more details.

Fixed-function AlphaTest

```
AlphaTest (Less | Greater | LEqual | GEqual | Equal | NotEqual | Always) CutoffV
```

Turns on fixed-function alpha testing. See documentation on alpha testing for more details.

Fixed-function Texture combiners

After the render state setup, use SetTexture commands to specify a number of Textures and their combining modes:

```
SetTexture textureProperty { combine options }
```

Details

Shader passes interact with Unity's rendering pipeline in several ways; for example, a Pass can indicate that it should only be used for deferred shading using the Tags command. Certain passes can also be executed multiple times on the same GameObject; for example, in forward rendering the "ForwardAdd" pass type is executed multiple times based on how many Lights are affecting the GameObject. See documentation on the Render Pipeline for more details.

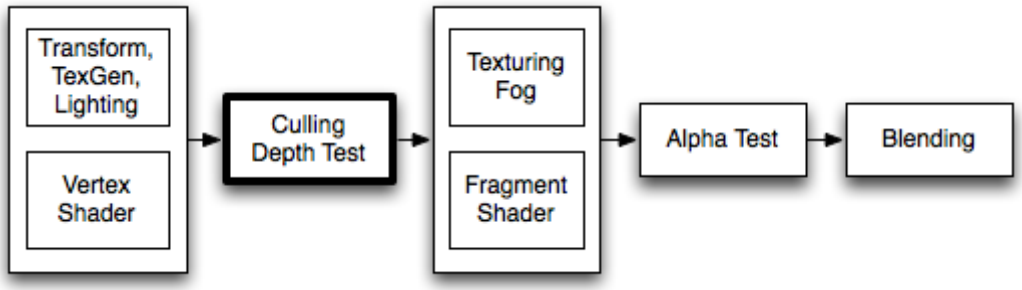
See also

There are several special passes available for reusing common functionality or implementing various high-end effects:

UsePass includes named passes from another shader.
GrabPass grabs the contents of the screen into a Texture, for use in a later pass.

ShaderLab: Culling & Depth Testing

Leave feedback



Culling is an optimization that does not render polygons facing away from the viewer. All polygons have a front and a back side. Culling makes use of the fact that most objects are closed; if you have a cube, you will never see the sides facing away from you (there is always a side facing you in front of it) so we don't need to draw the sides facing away. Hence the term: Backface culling.

The other feature that makes **rendering** looks correct is Depth testing. Depth testing makes sure that only the closest surfaces objects are drawn in a **scene**.

Syntax

Cull

```
Cull Back | Front | Off
```

Controls which sides of polygons should be culled (not drawn)

Back Don't render polygons facing away from the viewer *(default)*.

Front Don't render polygons facing towards the viewer. Used for turning objects inside-out.

Off Disables culling - all faces are drawn. Used for special effects.

ZWrite

```
ZWrite On | Off
```

Controls whether **pixels** from this object are written to the depth buffer (default is *On*). If you're drawng solid objects, leave this on. If you're drawing semitransparent effects, switch to `ZWrite Off`. For more details read below.

ZTest

```
ZTest Less | Greater | LEqual | GEqual | Equal | NotEqual | Always
```

How should depth testing be performed. Default is *LEqual* (draw objects in from or at the distance as existing objects; hide objects behind them).

Offset

```
Offset Factor, Units
```

Allows you specify a depth offset with two parameters. *factor* and *units*. *Factor* scales the maximum Z slope, with respect to X or Y of the polygon, and *units* scale the minimum resolvable depth buffer value. This allows you to force one polygon to be drawn on top of another although they are actually in the same position. For example `Offset 0, -1` pulls the polygon closer to the **camera** ignoring the polygon's slope, whereas `Offset -1, -1` will pull the polygon even closer when looking at a grazing angle.

Examples

This object will render only the backfaces of an object:

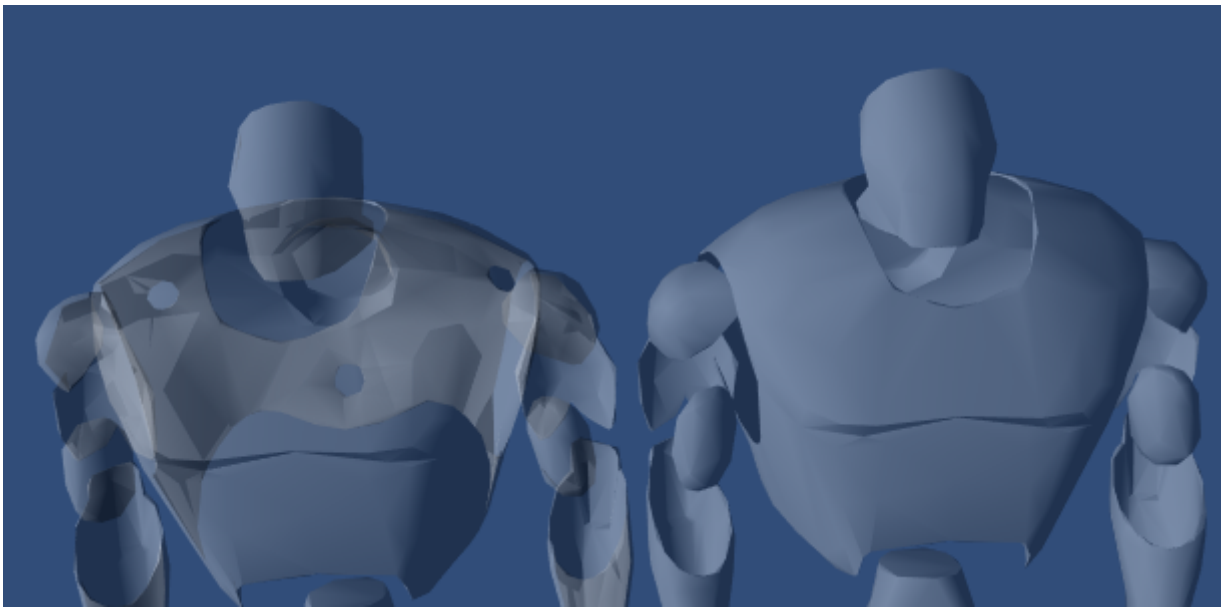
```
Shader "Show Insides" {
    SubShader {
        Pass {
            Material {
                Diffuse (1,1,1,1)
            }
            Lighting On
            Cull Front
        }
    }
}
```

Try to apply it to a cube, and notice how the geometry feels all wrong when you orbit around it. This is because you're only seeing the inside parts of the cube.

Transparent shader with depth writes

Usually semitransparent shaders do not write into the depth buffer. However, this can create draw order problems, especially with complex non-convex meshes. If you want to fade in & out meshes like that, then using a

shader that fills in the depth buffer before rendering transparency might be useful.



Semitransparent object; left: standard Transparent/Diffuse shader; right: shader that writes to depth buffer.

```
Shader "Transparent/Diffuse ZWrite" {
Properties {
    _Color ("Main Color", Color) = (1,1,1,1)
    _MainTex ("Base (RGB) Trans (A)", 2D) = "white" {}
}
SubShader {
    Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transpare
    LOD 200

    // extra pass that renders to depth buffer only
    Pass {
        ZWrite On
        ColorMask 0
    }

    // paste in forward rendering passes from Transparent/Diffuse
    UsePass "Transparent/Diffuse/FORWARD"
}
Fallback "Transparent/VertexLit"
}
```

Debugging Normals

The next one is more interesting; first we render the object with normal vertex lighting, then we render the backfaces in bright pink. This has the effects of highlighting anywhere your normals need to be flipped. If you see

physically-controlled objects getting 'sucked in' by any meshes, try to assign this shader to them. If any pink parts are visible, these parts will pull in anything unfortunate enough to touch it.

Here we go:

```
Shader "Reveal Backfaces" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" { }
    }
    SubShader {
        // Render the front-facing parts of the object.
        // We use a simple white material, and apply the main texture.
        Pass {
            Material {
                Diffuse (1,1,1,1)
            }
            Lighting On
            SetTexture [_MainTex] {
                Combine Primary * Texture
            }
        }

        // Now we render the back-facing triangles in the most
        // irritating color in the world: BRIGHT PINK!
        Pass {
            Color (1,0,1,1)
            Cull Front
        }
    }
}
```

Glass Culling

Controlling Culling is useful for more than debugging backfaces. If you have transparent objects, you quite often want to show the backfacing side of an object. If you render without any culling (**Cull Off**), you'll most likely have some rear faces overlapping some of the front faces.

Here is a simple shader that will work for convex objects (spheres, cubes, car windscreens).

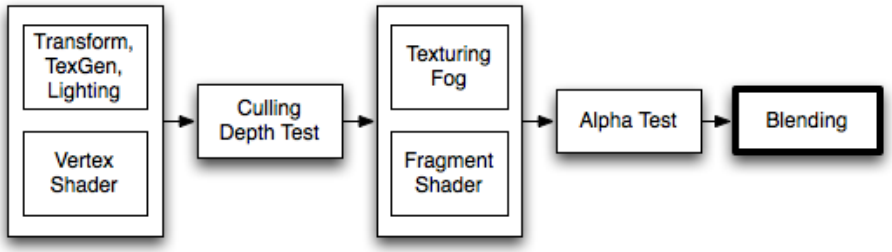
```
Shader "Simple Glass" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
```

```
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" { }
    }

    SubShader {
        // We use the material in many passes by defining them in the subshader.
        // Anything defined here becomes default values for all contained passes
        Material {
            Diffuse [_Color]
            Ambient [_Color]
            Shininess [_Shininess]
            Specular [_SpecColor]
            Emission [_Emission]
        }
        Lighting On
        SeparateSpecular On

        // Set up alpha blending
        Blend SrcAlpha OneMinusSrcAlpha

        // Render the back facing parts of the object.
        // If the object is convex, these will always be further away
        // than the front-faces.
        Pass {
            Cull Front
            SetTexture [_MainTex] {
                Combine Primary * Texture
            }
        }
        // Render the parts of the object facing us.
        // If the object is convex, these will be closer than the
        // back-faces.
        Pass {
            Cull Back
            SetTexture [_MainTex] {
                Combine Primary * Texture
            }
        }
    }
}
```

ShaderLab: Blending

Leave feedback

Blending is used to make transparent objects.



When graphics are rendered, after all **Shaders** have executed and all Textures have been applied, the **pixels** are written to the screen. How they are combined with what is already there is controlled by the Blend command.

Syntax

`Blend Off`: Turn off blending (this is the default)

`Blend SrcFactor DstFactor`: Configure and enable blending. The generated color is multiplied by the **SrcFactor**. The color already on screen is multiplied by **DstFactor** and the two are added together.

`Blend SrcFactor DstFactor, SrcFactorA DstFactorA`: Same as above, but use different factors for blending the alpha channel.

`BlendOp Op`: Instead of adding blended colors together, carry out a different operation on them.

`BlendOp OpColor, OpAlpha`: Same as above, but use different blend operation for color (RGB) and alpha (A) channels.

Additionally, you can set upper-rendertarget blending modes. When using multiple render target (MRT) rendering, the regular syntax above sets up the same blending modes for all render targets. The following syntax can set up different blending modes for individual render targets, where `N` is the render target index (0..7). This feature works on most modern APIs/GPUs (DX11/12, GLCore, Metal, PS4):

```
Blend N SrcFactor DstFactor
Blend N SrcFactor DstFactor, SrcFactorA DstFactorA
BlendOp N Op
BlendOp N OpColor, OpAlpha
```

`AlphaToMask On`: Turns on alpha-to-coverage. When MSAA is used, alpha-to-coverage modifies multisample coverage mask proportionally to the pixel Shader result alpha value. This is typically used for less aliased outlines than regular alpha test; useful for vegetation and other alpha-tested Shaders.

Blend operations

The following blend operations can be used:

Add	Add source and destination together.
Sub	Subtract destination from source.
RevSub	Subtract source from destination.
Min	Use the smaller of source and destination.
Max	Use the larger of source and destination.
LogicalClear	Logical operation: Clear (0) **DX11.1 only**.
LogicalSet	Logical operation: Set (1) **DX11.1 only**.
LogicalCopy	Logical operation: Copy (s) **DX11.1 only**.
LogicalCopyInverted	Logical operation: Copy inverted (!s) **DX11.1 only**.

LogicalNoop	Logical operation: Noop (d) **DX11.1 only**.
LogicalInvert	Logical operation: Invert (!d) **DX11.1 only**.
LogicalAnd	Logical operation: And (s & d) **DX11.1 only**.
LogicalNand	Logical operation: Nand !(s & d) **DX11.1 only**.
LogicalOr	Logical operation: Or (s | d) **DX11.1 only**.
LogicalNor	Logical operation: Nor !(s | d) **DX11.1 only**.
LogicalXor	Logical operation: Xor (s ^ d) **DX11.1 only**.
LogicalEquiv	Logical operation: Equivalence !(s ^ d) **DX11.1 only**.
LogicalAndReverse	Logical operation: Reverse And (s & !d) **DX11.1 only**.
LogicalAndInverted	Logical operation: Inverted And (!s & d) **DX11.1 only**.
LogicalOrReverse	Logical operation: Reverse Or (s | !d) **DX11.1 only**.
LogicalOrInverted	Logical operation: Inverted Or (!s | d) **DX11.1 only**.

Blend factors

All following properties are valid for both SrcFactor & DstFactor in the **Blend** command. **Source** refers to the calculated color, **Destination** is the color already on the screen. The blend factors are ignored if **BlendOp** is using logical operations.

One	The value of one - use this to let either the source or the destination color come through fully.
Zero	The value zero - use this to remove either the source or the destination values.
SrcColor	The value of this stage is multiplied by the source color value.
SrcAlpha	The value of this stage is multiplied by the source alpha value.
DstColor	The value of this stage is multiplied by frame buffer source color value.
DstAlpha	The value of this stage is multiplied by frame buffer source alpha value.
OneMinusSrcColor	The value of this stage is multiplied by (1 - source color).
OneMinusSrcAlpha	The value of this stage is multiplied by (1 - source alpha).
OneMinusDstColor	The value of this stage is multiplied by (1 - destination color).
OneMinusDstAlpha	The value of this stage is multiplied by (1 - destination alpha).

Details

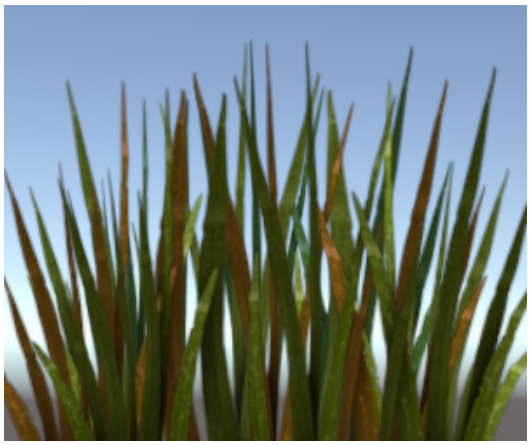
Below are the most common **blend types**:

```
Blend SrcAlpha OneMinusSrcAlpha // Traditional transparency
Blend One OneMinusSrcAlpha // Premultiplied transparency
Blend One One // Additive
Blend OneMinusDstColor One // Soft Additive
Blend DstColor Zero // Multiplicative
Blend DstColor SrcColor // 2x Multiplicative
```

Alpha blending, alpha testing, alpha-to-coverage

For drawing mostly fully opaque or fully transparent objects, where transparency is defined by the Texture's alpha channel (e.g. leaves, grass, chain fences etc.), several approaches are commonly used:

Alpha blending



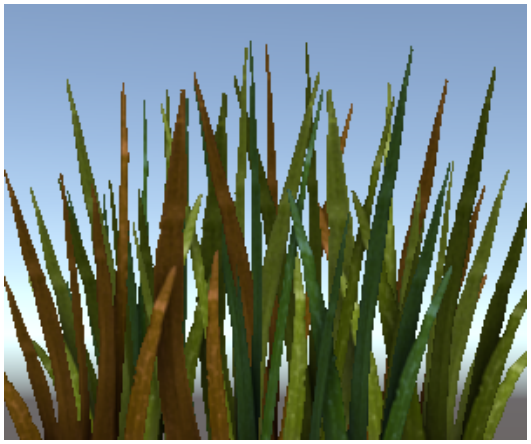
Regular alpha blending

This often means that objects have to be considered as “semitransparent”, and thus can’t use some of the **rendering** features (for example: deferred shading, can’t receive shadows). Concave or overlapping alpha-blended objects often also have draw ordering issues.

Often, alpha-blended Shaders also set transparent render queue, and turn off depth writes. So the Shader code looks like:

```
// inside SubShader
Tags { "Queue"="Transparent" "RenderType"="Transparent" "IgnoreProjector"="True" }

// inside Pass
ZWrite Off
Blend SrcAlpha OneMinusSrcAlpha
```

Alpha testing/cutout



clip() in pixel Shader

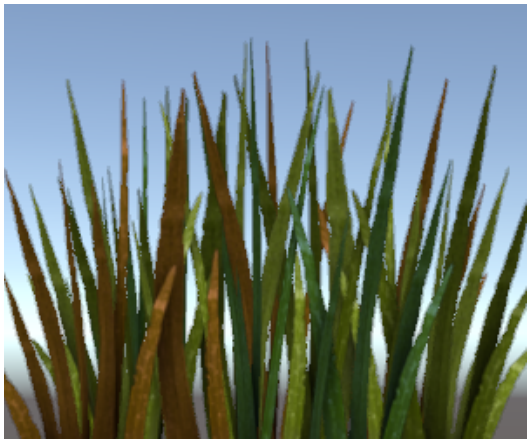
By using `clip()` HLSL instruction in the pixel Shader, a pixel can be “discarded” or not based on some criteria. This means that object can still be considered as fully opaque, and has no draw ordering issues. However, this means that all pixels are fully opaque or transparent, leading to aliasing (“jaggies”).

Often, alpha-tested Shaders also set cutout render queue, so the Shader code looks like this:

```
// inside SubShader
Tags { "Queue"="AlphaTest" "RenderType"="TransparentCutout" "IgnoreProjector"="True" }
```

```
// inside CGPROGRAM in the fragment Shader:
clip(textureColor.a - alphaCutoffValue);
```

Alpha-to-coverage



AlphaToMask On, at 4xMSAA

When using multisample anti-aliasing (MSAA, see QualitySettings), it is possible to improve the alpha testing approach by using alpha-to-coverage GPU functionality. This improves edge appearance, depending on the MSAA level used.

This functionality works best on texures that are mostly opaque or transparent, and have very thin “partially transparent” areas (grass, leaves and similar).

Often, alpha-to-coverage Shaders also set cutout render queue. So the Shader code looks like:

```
// inside SubShader
Tags { "Queue"="AlphaTest" "RenderType"="TransparentCutout" "IgnoreProjector"="True" }

// inside Pass
AlphaToMask On
```

Example

Here is a small example Shader that adds a Texture to whatever is on the screen already:

```
Shader "Simple Additive" {
    Properties {
        _MainTex ("Texture to blend", 2D) = "black" {}
    }
    SubShader {
        Tags { "Queue" = "Transparent" }
        Pass {
            Blend One One
            SetTexture [_MainTex] { combine texture }
        }
    }
}
```

ShaderLab: Pass Tags

Leave feedback

Passes use tags to tell how and when they expect to be rendered to the **rendering** engine.

Syntax

```
Tags { "TagName1" = "Value1" "TagName2" = "Value2" }
```

Specifies **TagName1** to have **Value1**, **TagName2** to have **Value2**. You can have as many tags as you like.

Details

Tags are basically key-value pairs. Inside a Pass tags are used to control which role this pass has in the lighting pipeline (ambient, vertex lit, pixel lit etc.) and some other options. Note that the following tags recognized by Unity *must* be inside Pass section and not inside **SubShader**!

LightMode tag

LightMode tag defines Pass' role in the lighting pipeline. See render pipeline for details. These tags are rarely used manually; most often **shaders** that need to interact with lighting are written as Surface Shaders and then all those details are taken care of.

Possible values for LightMode tag are:

Always: Always rendered; no lighting is applied.
ForwardBase: Used in Forward rendering, ambient, main directional light, vertex/SH lights and **lightmaps** are applied.
ForwardAdd: Used in Forward rendering; additive per-pixel lights are applied, one pass per light.
Deferred: Used in Deferred Shading; renders g-buffer.
ShadowCaster: Renders object depth into the shadowmap or a depth texture.
MotionVectors: Used to calculate per-object motion vectors.
PrepassBase: Used in legacy Deferred Lighting, renders normals and specular exponent.
PrepassFinal: Used in legacy Deferred Lighting, renders final color by combining textures, lighting and emission.
Vertex: Used in legacy Vertex Lit rendering when object is not lightmapped; all vertex lights are applied.
VertexLMRGBM: Used in legacy Vertex Lit rendering when object is lightmapped; on platforms where lightmap is RGBM encoded (PC & console).
VertexLM: Used in legacy Vertex Lit rendering when object is lightmapped; on platforms where lightmap is double-LDR encoded (mobile platforms).

PassFlags tag

A pass can indicate flags that change how rendering pipeline passes data to it. This is done by using **PassFlags** tag, with a value that is space-separated flag names. Currently the flags supported are:

OnlyDirectional: When used in ForwardBase pass type, this flag makes it so that only the main directional light and ambient/lightprobe data is passed into the shader. This means that data of non-

important lights is *not* passed into vertex-light or spherical harmonics shader variables. See Forward rendering for details.

RequireOptions tag

A pass can indicate that it should only be rendered when some external conditions are met. This is done by using **RequireOptions** tag, whose value is a string of space separated options. Currently the options supported by Unity are:

SoftVegetation: Render this pass only if Soft Vegetation is on in Quality Settings.

See Also

SubShaders can be given Tags as well, see SubShader Tags.

ShaderLab: Stencil

Leave feedback

The stencil buffer can be used as a general purpose per pixel mask for saving or discarding **pixels**.

The stencil buffer is usually an 8 bit integer per pixel. The value can be written to, increment or decremented. Subsequent draw calls can test against the value, to decide if a pixel should be discarded before running the pixel **shader**.

Syntax

Ref

```
Ref referenceValue
```

The value to be compared against (if Comp is anything else than *always*) and/or the value to be written to the buffer (if either Pass, Fail or ZFail is set to replace). 0–255 integer.

ReadMask

```
ReadMask readMask
```

An 8 bit mask as an 0–255 integer, used when comparing the reference value with the contents of the buffer (*referenceValue* & *readMask*) *comparisonFunction* (*stencilBufferValue* & *readMask*). Default: *255*.

WriteMask

```
WriteMask writeMask
```

An 8 bit mask as an 0–255 integer, used when writing to the buffer. Note that, like other write masks, it specifies which bits of stencil buffer will be affected by write (i.e. WriteMask 0 means that no bits are affected and not that 0 will be written). Default: *255*.

Comp

```
Comp comparisonFunction
```

The function used to compare the reference value to the current contents of the buffer. Default: *always*.

Pass

```
Pass stencilOperation
```

What to do with the contents of the buffer if the stencil test (and the depth test) passes. Default: *keep*.

Fail

```
Fail stencilOperation
```

What to do with the contents of the buffer if the stencil test fails. Default: *keep*.

ZFail

```
ZFail stencilOperation
```

What to do with the contents of the buffer if the stencil test passes, but the depth test fails. Default: *keep*.

Comp, Pass, Fail and ZFail will be applied to the front-facing geometry, unless *Cull Front* is specified, in which case it's back-facing geometry. You can also explicitly specify the two-sided stencil state by defining CompFront, PassFront, FailFront, ZFailFront (for front-facing geometry), and CompBack, PassBack, FailBack, ZFailBack (for back-facing geometry).

Comparison Function

Comparison function is one of the following:

Greater	Only render pixels whose reference value is greater than the value in the buffer.
GEqual	Only render pixels whose reference value is greater than or equal to the value in the buffer.
Less	Only render pixels whose reference value is less than the value in the buffer.
LEqual	Only render pixels whose reference value is less than or equal to the value in the buffer.
Equal	Only render pixels whose reference value equals the value in the buffer.
NotEqual	Only render pixels whose reference value differs from the value in the buffer.
Always	Make the stencil test always pass.
Never	Make the stencil test always fail.

Stencil Operation

Stencil operation is one of the following:

Keep	Keep the current contents of the buffer.
Zero	Write 0 into the buffer.

Replace Write the reference value into the buffer.
IncrSat Increment the current value in the buffer. If the value is 255 already, it stays at 255.
DecrSat Decrement the current value in the buffer. If the value is 0 already, it stays at 0.
Invert Negate all the bits.
IncrWrap Increment the current value in the buffer. If the value is 255 already, it becomes 0.
DecrWrap Decrement the current value in the buffer. If the value is 0 already, it becomes 255.

Deferred rendering path

Stencil functionality for objects rendered in the deferred **rendering path** is somewhat limited, as during the base pass and lighting pass the stencil buffer is used for other purposes. During those two stages stencil state defined in the shader will be ignored and only taken into account during the final pass. Because of that it's not possible to mask out these objects based on a stencil test, but they can still modify the buffer contents, to be used by objects rendered later in the frame. Objects rendered in the **forward rendering** path following the deferred path (e.g. transparent objects or objects without a surface shader) will set their stencil state normally again.

The deferred **rendering** path uses the three highest bits of the stencil buffer, plus up to four more highest bits - depending on how many light mask layers are used in the **scene**. It is possible to operate within the range of the "clean" bits using the stencil read and write masks, or you can force the **camera** to clean the stencil buffer after the lighting pass using **Camera**.clearStencilAfterLightingPass.

Example

The first example shader will write the value '2' wherever the depth test passes. The stencil test is set to always pass.

```
Shader "Red" {
    SubShader {
        Tags { "RenderType"="Opaque" "Queue"="Geometry"}
        Pass {
            Stencil {
                Ref 2
                Comp always
                Pass replace
            }

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            struct appdata {
                float4 vertex : POSITION;
            };
            struct v2f {
                float4 pos : SV_POSITION;
            };
            v2f vert(appdata v) {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                return o;
            }
```

```
            half4 frag(v2f i) : SV_Target {
                return half4(1,0,0,1);
            }
            ENDCG
        }
    }
}
```

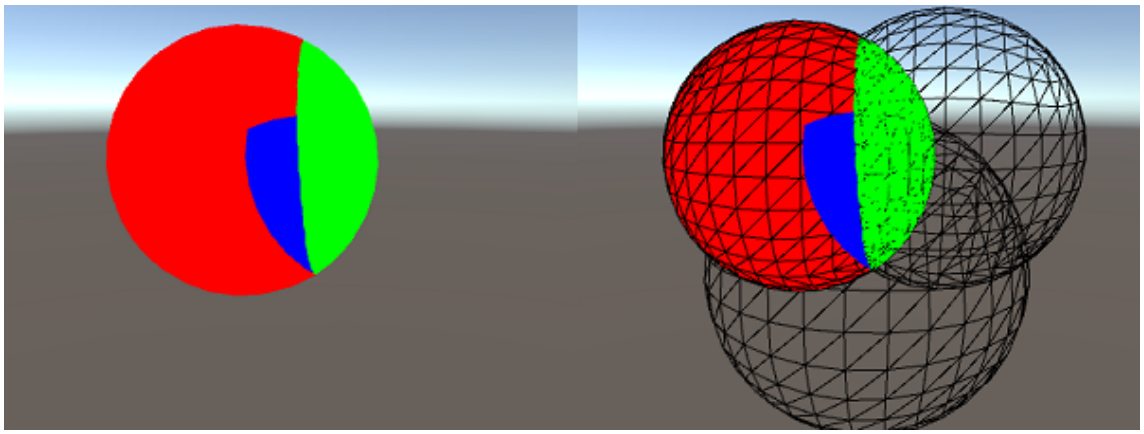
The second shader will pass only for the pixels which the first (red) shader passed, because it is checking for equality with the value ‘2’. It will also decrement the value in the buffer wherever it fails the Z test.

```
Shader "Green" {
    SubShader {
        Tags { "RenderType"="Opaque" "Queue"="Geometry+1"}
        Pass {
            Stencil {
                Ref 2
                Comp equal
                Pass keep
                ZFail decrWrap
            }

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            struct appdata {
                float4 vertex : POSITION;
            };
            struct v2f {
                float4 pos : SV_POSITION;
            };
            v2f vert(appdata v) {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                return o;
            }
            half4 frag(v2f i) : SV_Target {
                return half4(0,1,0,1);
            }
            ENDCG
        }
    }
}
```

The third shader will only pass wherever the stencil value is ‘1’, so only pixels at the intersection of both red and green spheres - that is, where the stencil is set to ‘2’ by the red shader and decremented to ‘1’ by the green shader.

```
Shader "Blue" {
    SubShader {
        Tags { "RenderType"="Opaque" "Queue"="Geometry+2"}
        Pass {
            Stencil {
                Ref 1
                Comp equal
            }

            CGPROGRAM
            #include "UnityCG.cginc"
            #pragma vertex vert
            #pragma fragment frag
            struct appdata {
                float4 vertex : POSITION;
            };
            struct v2f {
                float4 pos : SV_POSITION;
            };
            v2f vert(appdata v) {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                return o;
            }
            half4 frag(v2f i) : SV_Target {
                return half4(0,0,1,1);
            }
            ENDCG
        }
    }
}
```

The result:



Another example of a more directed effect. The sphere is first rendered with this shader to mark-up the proper regions in the stencil buffer:

```
Shader "HolePrepare" {
    SubShader {
        Tags { "RenderType"="Opaque" "Queue"="Geometry+1"}
        ColorMask 0
        ZWrite off
        Stencil {
            Ref 1
            Comp always
            Pass replace
        }

        CGINCLUDE
            struct appdata {
                float4 vertex : POSITION;
            };
            struct v2f {
                float4 pos : SV_POSITION;
            };
            v2f vert(appdata v) {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                return o;
            }
            half4 frag(v2f i) : SV_Target {
                return half4(1,1,0,1);
            }
        ENDCG

        Pass {
            Cull Front
            ZTest Less

            CGPROGRAM
            #pragma vertex vert
```

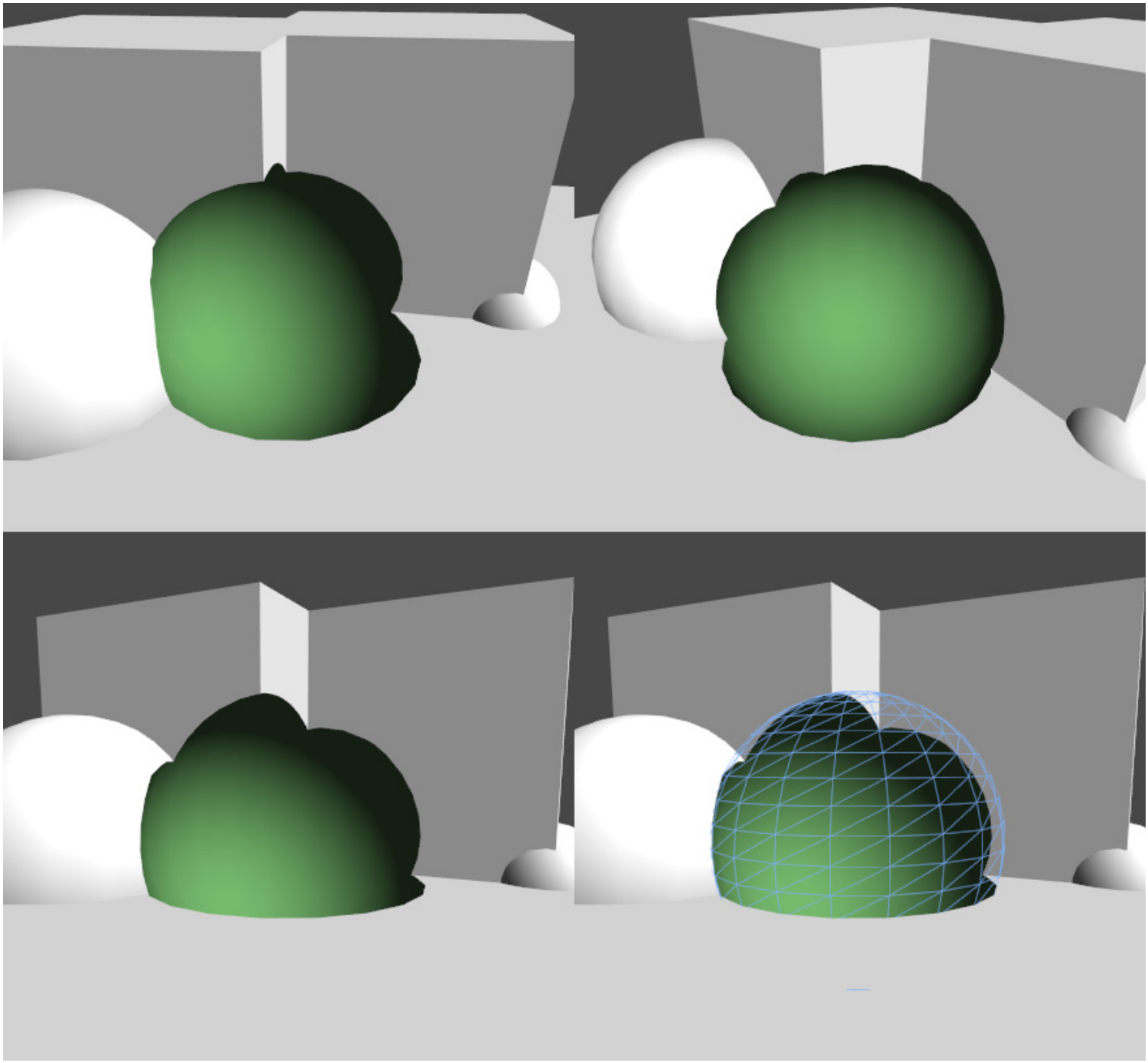
```
            #pragma fragment frag
            ENDCG
        }
        Pass {
            Cull Back
            ZTest Greater

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            ENDCG
        }
    }
}
```

And then rendered once more as a fairly standard **surface shader**, with the exception of front face culling, disabled depth test and stencil test discarding previously marked pixels:

```
Shader "Hole" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0)
    }
    SubShader {
        Tags { "RenderType"="Opaque" "Queue"="Geometry+2"}

        ColorMask RGB
        Cull Front
        ZTest Always
        Stencil {
            Ref 1
            Comp notequal
        }

        CGPROGRAM
        #pragma surface surf Lambert
        float4 _Color;
        struct Input {
            float4 color : COLOR;
        };
        void surf (Input IN, inout SurfaceOutput o) {
            o.Albedo = _Color.rgb;
            o.Normal = half3(0,0,-1);
            o.Alpha = 1;
        }
        ENDCG
```

```
        }
    }
```

The result:



ShaderLab: Name

Leave feedback

Syntax

```
Name "PassName"
```

Gives the *PassName* name to the current pass. Note that internally the names are turned to uppercase.

Details

A pass can be given a name so that a UsePass command can reference it.

ShaderLab: Legacy Lighting

Leave feedback

The material and lighting parameters are used to control the built-in vertex lighting. Vertex lighting is the standard Direct3D/OpenGL lighting model that is computed for each vertex. **Lighting on** turns it on. Lighting is affected by **Material** block, **ColorMaterial** and **SeparateSpecular** commands.

Note: *Material/Lighting commands have no effect when vertex programs are used; as in that case all calculations are completely described in the **shader**. It is advisable to use programmable shaders these days instead of legacy vertex lighting. For these you don't use any of the commands described here, instead you define your own vertex and fragment programs where you do all lighting, texturing and anything else yourself.*

Vertex Coloring & Lighting is the first effect to be calculated for any rendered geometry. It operates on the vertex level, and calculates the base color that is used before textures are applied.

Syntax

The top level commands control whether to use fixed function lighting or not, and some configuration options. The main setup is in the **Material Block**, detailed further below.

Color

```
Color color
```

Sets the object to a solid color. A color is either four RGBA values in parenthesis, or a color property name in square brackets.

Material

```
Material {Material Block}
```

The Material block is used to define the material properties of the object.

Lighting

```
Lighting On | Off
```

For the settings defined in the Material block to have any effect, you must enable Lighting with the *Lighting On* command. If lighting is off instead, the color is taken straight from the *Color* command.

SeparateSpecular

```
SeparateSpecular On | Off
```

This command makes specular lighting be added to the end of the shader pass, so specular lighting is unaffected by texturing. Only has effect when *Lighting On* is used.

ColorMaterial

```
ColorMaterial AmbientAndDiffuse | Emission
```

Makes per-vertex color be used instead of the colors set in the material. **AmbientAndDiffuse** replaces Ambient and Diffuse values of the material; **Emission** replaces Emission value of the material.

Material Block

This contains settings for how the material reacts to the light. Any of these properties can be left out, in which case they default to black (i.e. have no effect).

Diffuse color: The diffuse color **component**. This is an object's base color.

Ambient color: The ambient color component. This is the color the object has when it's hit by the ambient light set in the Lighting Window.

Specular color: The color of the object's specular highlight.

Shininess number: The sharpness of the highlight, between 0 and 1. At 0 you get a huge highlight that looks a lot like diffuse lighting, at 1 you get a tiny speck.

Emission color: The color of the object when it is not hit by any light.

The full color of lights hitting the object is:

Ambient * Lighting Window's Ambient Intensity setting + (Light Color * **Diffuse** + Light Color * **Specular**) + **Emission**

The light parts of the equation (within parenthesis) is repeated for all lights that hit the object.

Typically you want to keep the Diffuse and Ambient colors the same (all built-in Unity shaders do this).

Examples

Always render object in pure red:

```
Shader "Solid Red" {
    SubShader {
        Pass { Color (1,0,0,0) }
    }
}
```

Basic Shader that colors the object white and applies vertex lighting:

```
Shader "VertexLit White" {
    SubShader {
        Pass {
            Material {
                Diffuse (1,1,1,1)
                Ambient (1,1,1,1)
            }
            Lighting On
        }
    }
}
```

An extended version that adds material color as a property visible in Material **Inspector**:

```
Shader "VertexLit Simple" {
    Properties {
        _Color ("Main Color", COLOR) = (1,1,1,1)
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
                Ambient [_Color]
            }
            Lighting On
        }
    }
}
```

And finally, a full fledged vertex-lit shader (see also SetTexture reference page):

```
Shader "VertexLit" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
                Ambient [_Color]
                Shininess [_Shininess]
                Specular [_SpecColor]
                Emission [_Emission]
            }
            Lighting On
            SeparateSpecular On
            SetTexture [_MainTex] {
                Combine texture * primary DOUBLE, texture * primary
            }
        }
    }
}
```

ShaderLab: Legacy Texture Combiners

Leave feedback

After the basic vertex lighting has been calculated, textures are applied. In **ShaderLab** this is done using **SetTexture** command.

Note: *SetTexture commands have no effect when fragment programs are used; as in that case **pixel** operations are completely described in the **shader**. It is advisable to use programmable shaders these days instead of SetTexture commands.*

Fixed function texturing is the place to do old-style combiner effects. You can have multiple SetTexture commands inside a pass - all textures are applied in sequence, like layers in a painting program. SetTexture commands must be placed at the end of a Pass.

Syntax

```
SetTexture [TextureName] {Texture Block}
```

Assigns a texture. *TextureName* must be defined as a texture property. How to apply the texture is defined inside the *TextureBlock*.

The texture block controls how the texture is applied. Inside the texture block can be up to two commands: `combine` and `constantColor`.

Texture block `combine` command

`combine` *src1* * *src2*: Multiplies src1 and src2 together. The result will be darker than either input.

`combine` *src1* + *src2*: Adds src1 and src2 together. The result will be lighter than either input.

`combine` *src1* - *src2*: Subtracts src2 from src1.

`combine` *src1* `lerp` (*src2*) *src3*: Interpolates between src3 and src1, using the alpha of src2. Note that the **interpolation** is opposite direction: src1 is used when alpha is one, and src3 is used when alpha is zero.

`combine` *src1* * *src2* + *src3*: Multiplies src1 with the alpha component of src2, then adds src3.

All the **src** properties can be either one of *previous*, *constant*, *primary* or *texture*.

Previous is the the result of the previous SetTexture.
Primary is the color from the lighting calculation or the vertex color if it is bound.
Texture is the color of the texture specified by *TextureName* in the SetTexture (see above).
Constant is the color specified in **ConstantColor**.
Modifiers:

The formulas specified above can optionally be followed by the keywords **Double** or **Quad** to make the resulting color 2x or 4x as bright.

All the **src** properties, except `lerp` argument, can optionally be preceded by **one -** to make the resulting color negated.
All the **src** properties can be followed by **alpha** to take only the alpha channel.

Texture block `constantColor` command

ConstantColor color: Defines a constant color that can be used in the combine command.

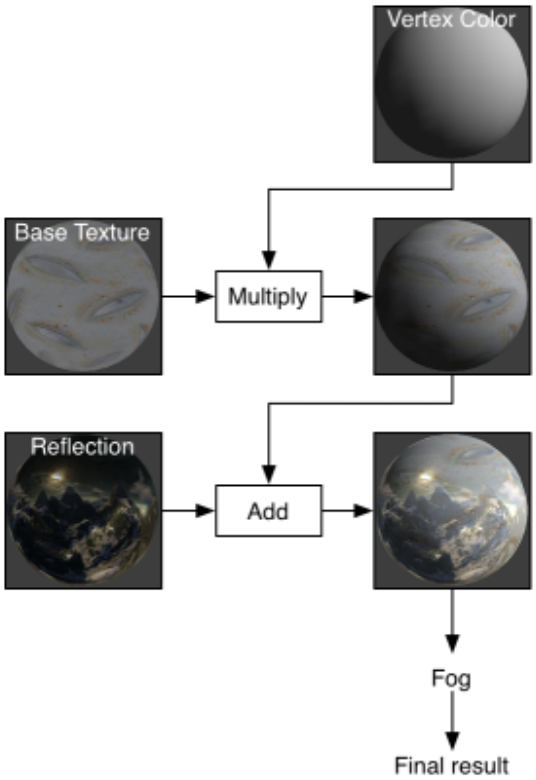
Functionality removed in Unity 5.0

Unity versions before 5.0 did support texture coordinate transformations with a `matrix` command inside a texture block. If you need this functionality now, consider rewriting your shader as a programmable shader instead, and do the UV transformation in the vertex shader.

Similarly, 5.0 removed signed add (`a+-b`), multiply signed add (`a*b+-c`), multiply subtract (`a*b-c`) and dot product (`dot3`, `dot3rgba`) texture combine modes. If you need them, do the math in the pixel shader instead.

Details

Before fragment programs existed, older graphics cards used a layered approach to textures. The textures are applied one after each other, modifying the color that will be written to the screen. For each texture, the texture is typically combined with the result of the previous operation. These days it is advisable to use actual fragment programs.



Note that each texture stage may or might not be clamped to 0..1 range, depending on the platform. This might affect SetTexture stages that can produce values higher than 1.0.

Separate Alpha & Color computation

By default, the combiner formula is used for calculating both the RGB and alpha component of the color. Optionally, you can specify a separate formula for the alpha calculation. This looks like this:

```
SetTexture [_MainTex] { combine previous * texture, previous + texture }
```

Here, we multiply the RGB colors and add the alpha.

Specular highlights

By default the **primary** color is the sum of the diffuse, ambient and specular colors (as defined in the Lighting calculation). If you specify **SeparateSpecular On** in the pass options, the specular color will be added in *after* the combiner calculation, rather than before. This is the default behavior of the built-in VertexLit shader.

Graphics hardware support

Modern graphics cards with fragment shader support ("shader model 2.0" on desktop, OpenGL ES 2.0 on mobile) support all **SetTexture** modes and at least 4 texture stages (many of them support 8). If you're running on really old hardware (made before 2003 on PC, or before iPhone3GS on mobile), you might have as low as two texture stages. The shader author should write separate SubShaders for the cards they want to support.

Examples

Alpha Blending Two Textures

This small examples takes two textures. First it sets the first combiner to just take the **_MainTex**, then is uses the alpha channel of **_BlendTex** to fade in the RGB colors of **_BlendTex**

```
Shader "Examples/2 Alpha Blended Textures" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _BlendTex ("Alpha Blended (RGBA) ", 2D) = "white" {}
    }
    SubShader {
        Pass {
            // Apply base texture
            SetTexture [_MainTex] {
                combine texture
            }
            // Blend in the alpha texture using the lerp operator
            SetTexture [_BlendTex] {
                combine texture lerp (texture) previous
            }
        }
    }
}
```

Alpha Controlled Self-illumination

This shader uses the alpha component of the **_MainTex** to decide where to apply lighting. It does this by applying the texture to two stages; In the first stage, the alpha value of the texture is used to blend between the vertex color and solid white. In the second stage, the RGB values of the texture are multiplied in.

```
Shader "Examples/Self-Illumination" {
    Properties {
        _MainTex ("Base (RGB) Self-Illumination (A)", 2D) = "white" {}
    }
    SubShader {
        Pass {
            // Set up basic white vertex lighting
            Material {
                Diffuse (1,1,1,1)
                Ambient (1,1,1,1)
            }
            Lighting On

            // Use texture alpha to blend up to white (= full illumination)
            SetTexture [_MainTex] {
                constantColor (1,1,1,1)
                combine constant lerp(texture) previous
            }
            // Multiply in texture
            SetTexture [_MainTex] {
                combine previous * texture
            }
        }
    }
}
```

We can do something else for free here, though; instead of blending to solid white, we can add a self-illumination color and blend to that. Note the use of **ConstantColor** to get a _SolidColor from the properties into the texture blending.

```
Shader "Examples/Self-Illumination 2" {
    Properties {
        _IlluminCol ("Self-Illumination color (RGB)", Color) = (1,1,1,1)
        _MainTex ("Base (RGB) Self-Illumination (A)", 2D) = "white" {}
    }
```

```
    SubShader {
        Pass {
            // Set up basic white vertex lighting
            Material {
                Diffuse (1,1,1,1)
                Ambient (1,1,1,1)
            }
            Lighting On

            // Use texture alpha to blend up to white (= full illumination)
            SetTexture [_MainTex] {
                // Pull the color property into this blender
                constantColor [_IlluminCol]
                // And use the texture's alpha to blend between it and
                // vertex color
                combine constant lerp(texture) previous
            }
            // Multiply in texture
            SetTexture [_MainTex] {
                combine previous * texture
            }
        }
    }
}
```

And finally, we take all the lighting properties of the vertexlit shader and pull that in:

```
Shader "Examples/Self-Illumination 3" {
    Properties {
        _IlluminCol ("Self-Illumination color (RGB)", Color) = (1,1,1,1)
        _Color ("Main Color", Color) = (1,1,1,0)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }

    SubShader {
        Pass {
            // Set up basic vertex lighting
            Material {
                Diffuse [_Color]
                Ambient [_Color]
                Shininess [_Shininess]
```

```
                Specular [_SpecColor]
                Emission [_Emission]
            }
            Lighting On

            // Use texture alpha to blend up to white (= full illumination)
            SetTexture [_MainTex] {
                constantColor [_IlluminCol]
                combine constant lerp(texture) previous
            }
            // Multiply in texture
            SetTexture [_MainTex] {
                combine previous * texture
            }
        }
    }
}
```

ShaderLab: Legacy Alpha Testing

Leave feedback

The alpha test is a last chance to reject a **pixel** from being written to the screen.

Note: *AlphaTest commands have no effect when fragment programs are used; on most platforms alpha testing is done in the **shader** using HLSL clip() function. It is advisable to use programmable shaders these days instead of SetTexture commands.*

After the final output color has been calculated, the color can optionally have its alpha value compared to a fixed value. If the test fails, the pixel is not written to the display.

Syntax

```
AlphaTest Off
```

Render all pixels (default) or...

```
AlphaTest comparison AlphaValue
```

Set up the alpha test to only render pixels whose alpha value is within a certain range.

Comparison

Comparison is one of the following words:

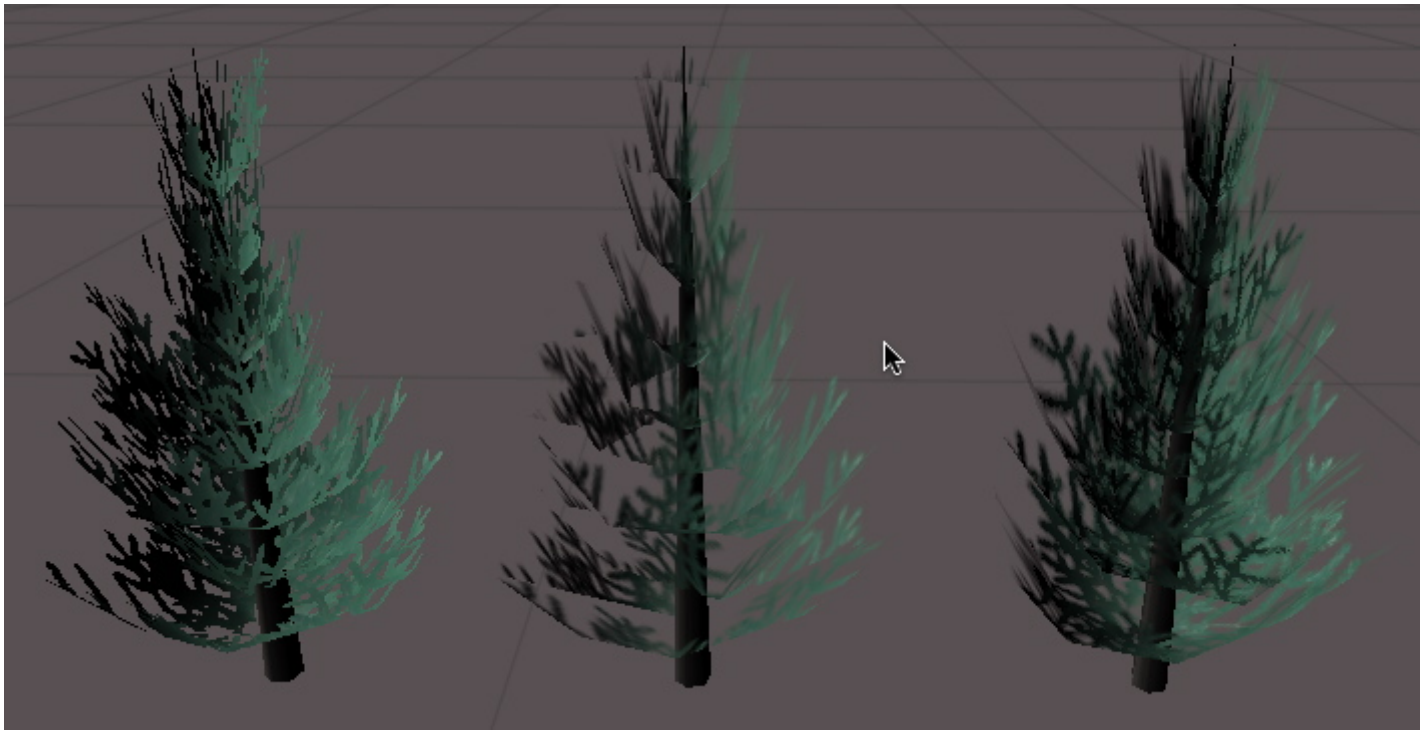
Greater	Only render pixels whose alpha is greater than *AlphaValue*.
GEqual	Only render pixels whose alpha is greater than or equal to *AlphaValue*.
Less	Only render pixels whose alpha value is less than *AlphaValue*.
LEqual	Only render pixels whose alpha value is less than or equal to from *AlphaValue*.
Equal	Only render pixels whose alpha value equals *AlphaValue*.
NotEqual	Only render pixels whose alpha value differs from *AlphaValue*.
Always	Render all pixels. This is functionally equivalent to *AlphaTest Off*.
Never	Don't render any pixels.

AlphaValue

A floating-point number between 0 and 1. This can also be a variable reference to a float or range property, in which case it should be written using the standard square bracket notation (*[VariableName]*).

Details

The alpha test is important when **rendering** concave objects with transparent parts. The graphics card maintains a record of the depth of every pixel written to the screen. If a new pixel is further away than one already rendered, the new pixel is not written to the display. This means that even with Blending, objects will not show through.



In this figure, the tree on the left is rendered using **AlphaTest**. Note how the pixels in it are either completely transparent or opaque. The center tree is rendered using only **Alpha Blending** - notice how transparent parts of nearby branches cover the distant leaves because of the depth buffer. The tree on the right is rendered using the last example shader - which implements a combination of blending and alpha testing to hide any artifacts.

Examples

The simplest possible example, assign a texture with an alpha channel to it. The object will only be visible where alpha is greater than 0.5

```
Shader "Simple Alpha Test" {
    Properties {
        _MainTex ("Base (RGB) Transparency (A)", 2D) = "" {}
    }
    SubShader {
        Pass {
            // Only render pixels with an alpha larger than 50%
            AlphaTest Greater 0.5
            SetTexture [_MainTex] { combine texture }
        }
    }
}
```

This is not much good by itself. Let us add some lighting and make the cutoff value tweakable:

```
Shader "Cutoff Alpha" {
    Properties {
        _MainTex ("Base (RGB) Transparency (A)", 2D) = "" {}
        _Cutoff ("Alpha cutoff", Range (0,1)) = 0.5
    }
    SubShader {
        Pass {
            // Use the Cutoff parameter defined above to determine
            // what to render.
            AlphaTest Greater [_Cutoff]
            Material {
                Diffuse (1,1,1,1)
                Ambient (1,1,1,1)
            }
            Lighting On
            SetTexture [_MainTex] { combine texture * primary }
        }
    }
}
```

When rendering plants and trees, many games have the hard edges typical of alpha testing. A way around that is to render the object twice. In the first pass, we use alpha testing to only render pixels that are more than 50% opaque. In the second pass, we alpha-blend the graphic in the parts that were cut away, without recording the depth of the pixel. We might get a bit of confusion as further away branches overwrite the nearby ones, but in practice, that is hard to see as leaves have a lot of visual detail in them.

```
Shader "Vegetation" {
    Properties {
        _Color ("Main Color", Color) = (.5, .5, .5, .5)
        _MainTex ("Base (RGB) Alpha (A)", 2D) = "white" {}
        _Cutoff ("Base Alpha cutoff", Range (0,.9)) = .5
    }
    SubShader {
        // Set up basic lighting
        Material {
            Diffuse [_Color]
            Ambient [_Color]
        }
        Lighting On
```

```
        // Render both front and back facing polygons.
        Cull Off

        // first pass:
        // render any pixels that are more than [_Cutoff] opaque
        Pass {
            AlphaTest Greater [_Cutoff]
            SetTexture [_MainTex] {
                combine texture * primary, texture
            }
        }

        // Second pass:
        // render in the semitransparent details.
        Pass {
            // Dont write to the depth buffer
            ZWrite off
            // Don't write pixels we have already written.
            ZTest Less
            // Only render pixels less or equal to the value
            AlphaTest LEqual [_Cutoff]

            // Set up alpha blending
            Blend SrcAlpha OneMinusSrcAlpha

            SetTexture [_MainTex] {
                combine texture * primary, texture
            }
        }
    }
}
```

Note that we have some setup inside the **SubShader**, rather than in the individual passes. Any state set in the **SubShader** is inherited as defaults in passes inside it.

ShaderLab: Legacy Fog

Leave feedback

Fog parameters are controlled with Fog command.

Fogging blends the color of the generated **pixels** down towards a constant color based on distance from **camera**. Fogging does not modify a blended pixel's alpha value, only its RGB **components**.

Syntax

Fog

```
Fog {Fog Commands}
```

Specify fog commands inside curly braces.

Mode

```
Mode Off | Global | Linear | Exp | Exp2
```

Defines fog mode. Default is global, which translates to Off or Exp2 depending whether fog is turned on in Render Settings.

Color

```
Color ColorValue
```

Sets fog color.

Density

```
Density FloatValue
```

Sets density for exponential fog.

Range

```
Range FloatValue, FloatValue
```

Sets near & far range for linear fog.

Details

Default fog settings are based on settings in the Lighting Window: fog mode is either **Exp2** or **Off**; density & color taken from settings as well.

Note that if you use fragment programs, Fog settings of the **shader** will still be applied. On platforms where there is no fixed function Fog functionality, Unity will patch shaders at runtime to support the requested Fog mode.

ShaderLab: Legacy BindChannels

Leave feedback

BindChannels command allows you to specify how vertex data maps to the graphics hardware.

Note: *_BindChannels has no effect when vertex programs are used, as in that case bindings are controlled by vertex shader inputs. It is advisable to use programmable shaders these days instead of fixed function vertex processing.*

By default, Unity figures out the bindings for you, but in some cases you want custom ones to be used.

For example you could map the primary UV set to be used in the first texture stage and the secondary UV set to be used in the second texture stage; or tell the hardware that vertex colors should be taken into account.

Syntax

```
BindChannels { Bind "source", target }
```

Specifies that vertex data *source* maps to hardware *target*.

Source can be one of:

Vertex: vertex position
Normal: vertex normal
Tangent: vertex tangent
Texcoord: primary UV coordinate
Texcoord1: secondary UV coordinate
Color: per-vertex color
Target can be one of:

Vertex: vertex position
Normal: vertex normal
Tangent: vertex tangent
Texcoord0, **Texcoord1**, ...: texture coordinates for corresponding texture stage
Texcoord: texture coordinates for all texture stages
Color: vertex color

Details

Unity places some restrictions on which sources can be mapped to which targets. Source and target must match for **Vertex**, **Normal**, **Tangent** and **Color**. Texture coordinates from the **mesh** (**Texcoord** and **Texcoord1**) can be mapped into texture coordinate targets (**Texcoord** for all texture stages, or **TexcoordN** for a specific stage).

There are two typical use cases for BindChannels:

Shaders that take vertex colors into account.
Shaders that use two UV sets.

Examples

```
// Maps the first UV set to the first texture stage
// and the second UV set to the second texture stage
BindChannels {
   Bind "Vertex", vertex
   Bind "texcoord", texcoord0
   Bind "texcoord1", texcoord1
}
```

```
// Maps the first UV set to all texture stages
// and uses vertex colors
BindChannels {
   Bind "Vertex", vertex
   Bind "texcoord", texcoord
   Bind "Color", color
}
```

ShaderLab: UsePass

Leave feedback

The UsePass command uses named passes from another **shader**.

Syntax

```
UsePass "Shader/Name"
```

Inserts all passes with a given name from a given shader. *Shader/Name* contains the name of the shader and the name of the pass, separated by a slash character. Note that only first supported subshader is taken into account.

Details

Some of the shaders could reuse existing passes from other shaders, reducing code duplication. For example, you might have a shader pass that draws object outline, and you'd want to reuse that pass in other shaders. The UsePass command does just that - it includes a given pass from another shader. As an example the following command uses the pass with the name "SHADOWCASTER" from the built-in *VertexLit* shader:

```
UsePass "VertexLit/SHADOWCASTER"
```

In order for UsePass to work, a name must be given to the pass one wishes to use. The Name command inside the pass gives it a name:

```
Name "MyPassName"
```

Note that internally all pass names are uppercased, so UsePass must refer to the name **in uppercase**.

ShaderLab: GrabPass

Leave feedback

GrabPass is a special pass type - it grabs the contents of the screen where the object is about to be drawn into a texture. This texture can be used in subsequent passes to do advanced image based effects.

Syntax

The GrabPass belongs inside a subshader. It can take two forms:

Just `GrabPass { }` grabs the current screen contents into a texture. The texture can be accessed in further passes by `_GrabTexture` name. Note: this form of grab pass will do the time-consuming screen grabbing operation for each object that uses it.
`GrabPass { "TextureName" }` grabs the current screen contents into a texture, but will only do that once per frame for the first object that uses the given texture name. The texture can be accessed in further passes by the given texture name. This is a more performant method when you have multiple objects using GrabPass in the **scene**.
Additionally, GrabPass can use Name and Tags commands.

Example

Here is an inefficient way to invert the colors of what was rendered before:

```
Shader "GrabPassInvert"
{
    SubShader
    {
        // Draw ourselves after all opaque geometry
        Tags { "Queue" = "Transparent" }

        // Grab the screen behind the object into _BackgroundTexture
        GrabPass
        {
            "_BackgroundTexture"
        }

        // Render the object with the texture generated above, and invert the colors
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct v2f
            {
                float4 grabPos : TEXCOORD0;
                float4 pos : SV_POSITION;
            };

            v2f vert(appdata_base v) {
```

```
                v2f o;
                // use UnityObjectToClipPos from UnityCG.cginc to calculate
                // the clip-space of the vertex
                o.pos = UnityObjectToClipPos(v.vertex);
                // use ComputeGrabScreenPos function from UnityCG.cginc
                // to get the correct texture coordinate
                o.grabPos = ComputeGrabScreenPos(o.pos);
                return o;
            }

            sampler2D _BackgroundTexture;

            half4 frag(v2f i) : SV_Target
            {
                half4 bgcolor = tex2Dproj(_BackgroundTexture, i.grabPos);
                return 1 - bgcolor;
            }
            ENDCG
        }

    }
}
```

This **shader** has two passes: The first pass grabs whatever is behind the object at the time of **rendering**, then applies that in the second pass. Note that the same effect could be achieved more efficiently using an invert blend mode.

See Also

Regular Pass command.
Platform differences.
Built-in shader helper functions.

ShaderLab: SubShader Tags

Leave feedback

Subshaders use tags to tell how and when they expect to be rendered to the **rendering** engine.

Syntax

```
Tags { "TagName1" = "Value1" "TagName2" = "Value2" }
```

Specifies **TagName1** to have **Value1**, **TagName2** to have **Value2**. You can have as many tags as you like.

Details

Tags are basically key-value pairs. Inside a SubShader tags are used to determine rendering order and other parameters of a subshader. Note that the following tags recognized by Unity **must** be inside SubShader section and not inside Pass!

In addition to built-in tags recognized by Unity, you can use your own tags and query them using Material.GetTag function.

Rendering Order - Queue tag

You can determine in which order your objects are drawn using the *Queue* tag. A **Shader** decides which render queue its objects belong to, this way any Transparent shaders make sure they are drawn after all opaque objects and so on.

There are four pre-defined render queues, but there can be more queues in between the predefined ones. The predefined queues are:

`Background` - this render queue is rendered before any others. You'd typically use this for things that really need to be in the background.
`Geometry` *(default)* - this is used for most objects. Opaque geometry uses this queue.
`AlphaTest` - alpha tested geometry uses this queue. It's a separate queue from `Geometry` one since it's more efficient to render alpha-tested objects after all solid ones are drawn.
`Transparent` - this render queue is rendered after *Geometry* and `AlphaTest`, in back-to-front order. Anything alpha-blended (i.e. shaders that don't write to depth buffer) should go here (glass, particle effects).
`Overlay` - this render queue is meant for overlay effects. Anything rendered last should go here (e.g. lens flares).

```
Shader "Transparent Queue Example"
{
     SubShader
     {
        Tags { "Queue" = "Transparent" }
```

```
        Pass
        {
            // rest of the shader body...
        }
    }
}
```

An example illustrating how to render something in the transparent queue

For special uses in-between queues can be used. Internally each queue is represented by integer index; `Background` is 1000, `Geometry` is 2000, `AlphaTest` is 2450, `Transparent` is 3000 and `Overlay` is 4000. If a shader uses a queue like this:

```
Tags { "Queue" = "Geometry+1" }
```

This will make the object be rendered after all opaque objects, but before transparent objects, as render queue index will be 2001 (geometry plus one). This is useful in situations where you want some objects be always drawn between other sets of objects. For example, in most cases transparent water should be drawn after opaque objects but before transparent objects.

Queues up to 2500 ("Geometry+500") are consided "opaque" and optimize the drawing order of the objects for best performance. Higher rendering queues are considered for "transparent objects" and sort objects by distance, starting rendering from the furthest ones and ending with the closest ones. Skyboxes are drawn in between all opaque and all transparent objects.

RenderType tag

`RenderType` tag categorizes shaders into several predefined groups, e.g. is is an opaque shader, or an alpha-tested shader etc. This is used by Shader Replacement and in some cases used to produce camera's depth texture.

DisableBatching tag

Some shaders (mostly ones that do object-space vertex deformations) do not work when Draw Call Batching is used – that's because batching transforms all geometry into world space, so "object space" is lost.

`DisableBatching` tag can be used to indicate that. There are three possible values: "True" (always disables batching for this shader), "False" (does not disable batching; this is default) and "LODFading" (disable batching when LOD fading is active; mostly used on trees).

ForceNoShadowCasting tag

If `ForceNoShadowCasting` tag is given and has a value of “True”, then an object that is rendered using this subshader will never cast shadows. This is mostly useful when you are using shader replacement on transparent objects and you do not wont to inherit a shadow pass from another subshader.

IgnoreProjector tag

If `IgnoreProjector` tag is given and has a value of “True”, then an object that uses this shader will not be affected by Projectors. This is mostly useful on semitransparent objects, because there is no good way for Projectors to affect them.

CanUseSpriteAtlas tag

Set `CanUseSpriteAtlas` tag to “False” if the shader is meant for **sprites**, and will not work when they are packed into atlases (see Sprite Packer).

PreviewType tag

`PreviewType` indicates how the material inspector preview should display the material. By default materials are displayed as spheres, but PreviewType can also be set to “Plane” (will display as 2D) or “Skybox” (will display as skybox).

See Also

Passes can be given Tags as well, see Pass Tags.

ShaderLab: Fallback

Leave feedback

After all **Subshaders** a Fallback can be defined. It basically says “if none of **subshaders** can run on this hardware, try using the ones from another **shader**”.

Syntax

```
Fallback "name"
```

Fallback to shader with a given *name* or...

```
Fallback Off
```

Explicitly state that there is no fallback and no warning should be printed, even if no subshaders can run on this hardware.

Details

A fallback statement has the same effect as if all subshaders from the other shader would be inserted into its place.

Example

```
Shader "example" {
    // properties and subshaders here...
    Fallback "otherexample"
}
```

ShaderLab: CustomEditor

Leave feedback

A CustomEditor can be defined for your **shader**. When you do this Unity will look for a class that extends ShaderGUI with this name. If one is found any material that uses this **shader** will use this ShaderGUI. See Custom Shader GUI for examples.

Syntax

```
    CustomEditor "name"
```

Use the ShaderGUI with a given *name*.

Details

A CustomEditor statement effects all materials that use this Shader

Example

```
Shader "example" {
    // properties and subshaders here...
    CustomEditor "MyCustomEditor"
}
```

ShaderLab: other commands

Leave feedback

Category

Category is a logical grouping of any commands below it. This is mostly used to "inherit" **rendering** state. For example, your **shader** might have multiple subshaders, and each of them requires fog to be off, blending set to additive, **etc**. You can use Category for that:

```
Shader "example" {
Category {
    Fog { Mode Off }
    Blend One One
    SubShader {
        // ...
    }
    SubShader {
        // ...
    }
    // ...
}
}
```

Category block only affects shader parsing, it's exactly the same as "pasting" any state set inside Category into all blocks below it. It does not affect shader execution speed at all.

Shader assets

Leave feedback

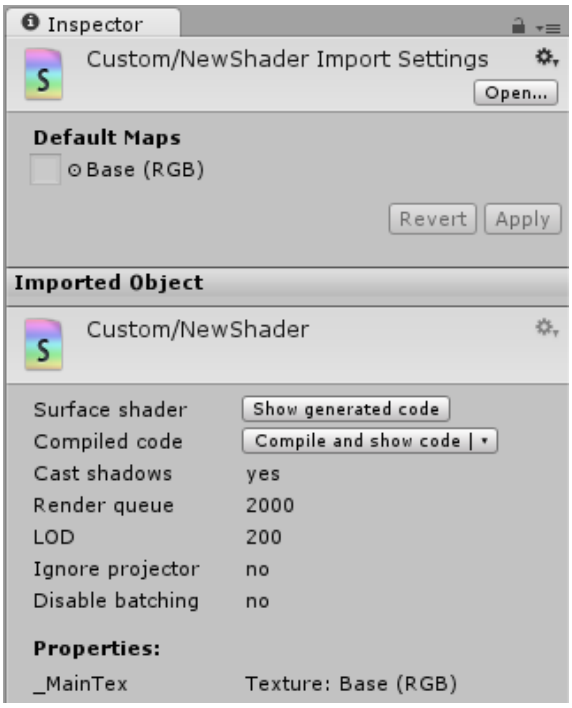
SWITCH TO SCRIPTING

Shaders are assets that contain code and instructions for the graphics card to execute. Materials reference shaders, and setup their parameters (textures, colors and so on).

Unity contains some built-in shaders that are always available in your project (for example, the Standard shader). You can also write your own shaders and apply post-processing effects.

Creating a new shader

To create a new Shader, use **Assets** > **Create** > **Shader** from the main menu or the **Project View** context menu. A shader is a text file similar to a C# script, and is written in a combination of Cg/HLSL and **ShaderLab** languages (see writing shaders page for details).



Shader inspector.

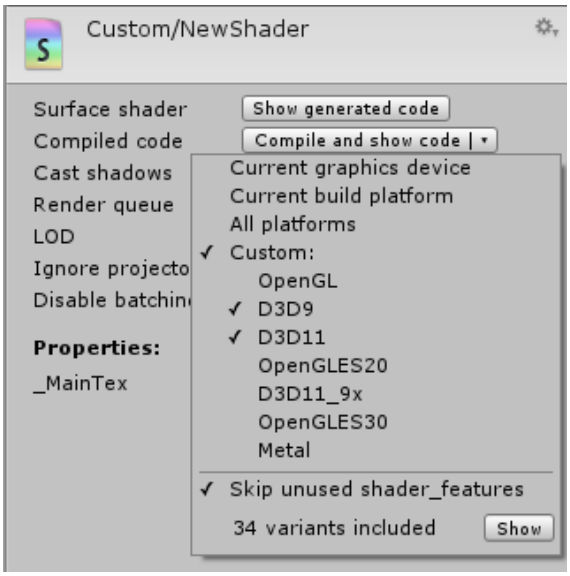
Shader import settings

This **inspector** section allows specifying default textures for a shader. Whenever a new Material is created with this shader, these textures are automatically assigned.

Shader Inspector

The Shader Inspector displays basic information about the shader (mostly shader tags), and allows compiling and inspecting low-level compiled code.

For Surface Shaders, the **Show generated code** button displays all the code that Unity generates to handle lighting and shadowing. If you really want to customize the generated code, you can just copy and paste all of it back to your original shader file and start tweaking.



Shader compilation popup menu.

The pop-up menu of the **Compile and show code** button allows inspecting final compiled shader code (e.g. assembly on Direct3D9, or low-level optimized GLSL for OpenGL ES) for selected platforms. This is mostly useful while optimizing shaders for performance; often you do want to know how many low-level instructions here generated in the end.

The low-level generated code is useful for pasting into GPU shader performance analysis tools (like AMD GPU ShaderAnalyzer or PVRShaderEditor).

Shader compilation details

On shader import time, Unity does not compile the whole shader. This is because majority of shaders have a lot of variants inside, and compiling all of them, for all possible platforms, would take a very long time. Instead, this is done:

At import time, only do minimal processing of the shader (surface shader generation etc.).
Actually compile the shader variants only when needed.
Instead of typical work of compiling 100–10000 internal shaders at import time, this usually ends up compiling just a handful.
At player build time, all the "not yet compiled" shader variants are compiled, so that they are in the game data even if the editor did not happen to use them.

However, this does mean that a shader might have an error in there, which is not detected at shader import time. For example, you're running editor using Direct3D 11, but a shader has an error if compiled for OpenGL. Or some variants of the shader does not fit into shader model 2.0 instruction limits, **etc**. These errors will be shown in the inspector if editor needs them; but it's also a good practice to manually fully compile the shader for platforms you need, to check for errors. This can be done using the **Compile and show code** pop-up menu in the shader inspector.

Shader compilation is carried out using a background process named `UnityShaderCompiler` that is started by Unity whenever it needs to compile shaders. Multiple compiler processes can be started (generally one per CPU core in your machine), so that at player build time shader compilation can be done in parallel. While the editor is not compiling shaders, the compiler processes do nothing and do not consume computer resources, so there's no need to worry about them. They are also shut down when Unity editor quits.

Individual shader variant compilation results are cached in the project, under `Library/ShaderCache` folder. This means that 100% identical shaders or their snippets will reuse previously compiled results. It also means that the shader cache folder can become quite large, if you have a lot of shaders that are changed often. It is always safe to delete it; it will just cause shader variants to be recompiled.

Further reading

Material assets.
Writing Shaders overview.
Shader reference.

Advanced ShaderLab topics

Leave feedback

Read those to improve your ShaderLab-fu!

Unity's Rendering Pipeline

Leave feedback

Shaders define both how an object looks by itself (its material properties) and how it reacts to the light. Because lighting calculations must be built into the **shader**, and there are many possible light & shadow types, writing quality shaders that "just work" would be an involved task. To make it easier, Unity has Surface Shaders, where all the lighting, shadowing, lightmapping, forward vs. deferred **rendering** things are taken care of automatically.

This document describes the pecularities of Unity's lighting & rendering pipeline and what happens behind the **scenes** of Surface Shaders.

Rendering Paths

How lighting is applied and which Passes of the shader are used depends on which Rendering Path is used. Each pass in a shader communicates its lighting type via Pass Tags.

In Forward Rendering, `ForwardBase` and `ForwardAdd` passes are used.
In Deferred Shading, `Deferred` pass is used.
In legacy Deferred Lighting, `PrepassBase` and `PrepassFinal` passes are used.
In legacy Vertex Lit, `Vertex`, `VertexLMRGBM` and `VertexLM` passes are used.
In any of the above, to render Shadows or a depth texture, `ShadowCaster` pass is used.

Forward Rendering path

`ForwardBase` pass renders ambient, lightmaps, main directional light and not important (vertex/SH) lights at once. `ForwardAdd` pass is used for any additive per-pixel lights; one invocation per object illuminated by such light is done. See Forward Rendering for details.

If forward rendering is used, but a shader does not have forward-suitable passes (i.e. neither `ForwardBase` nor `ForwardAdd` pass types are present), then that object is rendered just like it would in Vertex Lit path, see below.

Deferred Shading path

`Deferred` pass renders all information needed for lighting (in built-in shaders: diffuse color, specular color, smoothness, world space normal, emission). It also adds lightmaps, reflection probes and ambient lighting into the emission channel. See Deferred Shading for details.

Legacy Deferred Lighting path

`PrepassBase` pass renders normals & specular exponent; `PrepassFinal` pass renders final color by combining textures, lighting & emissive material properties. All regular in-scene lighting is done separately in screen-space. See Deferred Lighting for details.

Legacy Vertex Lit Rendering path

Since vertex lighting is most often used on platforms that do not support programmable shaders, Unity can't create multiple shader variants internally to handle lightmapped vs. non-lightmapped cases. So to handle lightmapped and non-lightmapped objects, multiple passes have to be written explicitly.

`Vertex` pass is used for non-lightmapped objects. All lights are rendered at once, using a fixed function OpenGL/Direct3D lighting model (Blinn-Phong)
`VertexLMRGBM` pass is used for lightmapped objects, when lightmaps are RGBM encoded (PC and consoles). No realtime lighting is applied; pass is expected to combine textures with a lightmap.

`VertexLMM` pass is used for lightmapped objects, when lightmaps are double-LDR encoded (mobile platforms). No realtime lighting is applied; pass is expected to combine textures with a lightmap.

See Also

Graphics Command Buffers for how to extend Unity's rendering pipeline.

Performance tips when writing shaders

Leave feedback

Only compute what you need

The more computations and processing your **shader** code needs to do, the more it will impact the performance of your game. For example, supporting color per material is nice to make a **shader** more flexible, but if you always leave that color set to white then useless computations are performed for each vertex or **pixel** rendered on screen.

The frequency of computations will also impact the performance of your game. Usually there are many more pixels rendered (and subsequently more pixel shader executions) than there are vertices (vertex shader executions), and more vertices than objects being rendered. Where possible, move computations out of the pixel shader code into the the **vertex shader** code, or move them out of shaders completely and set the values in a script.

Optimized Surface Shaders

Surface Shaders are great for writing shaders that interact with lighting. However, their default options are tuned to cover a broad number of general cases. Tweak these for specific situations to make shaders run faster or at least be smaller:

The `approxview` directive for shaders that use view direction (i.e. Specular) makes the view direction normalized per vertex instead of per pixel. This is approximate, but often good enough.
The `halfasview` for Specular shader types is even faster. The half-vector (halfway between lighting direction and view vector) is computed and normalized per vertex, and the lighting function receives the half-vector as a parameter instead of the view vector.
`noforwardadd` makes a shader fully support one-directional light in **Forward rendering** only. The rest of the lights can still have an effect as per-vertex lights or spherical harmonics. This is great to make your shader smaller and make sure it always renders in one pass, even with multiple lights present.
`noambient` disables ambient lighting and spherical harmonics lights on a shader. This can make performance slightly faster.

Precision of computations

When writing shaders in Cg/HLSL, there are three basic number types: `float`, `half` and `fixed` (see Data Types and Precision).

For good performance, always use the lowest precision that is possible. This is especially important on mobile platforms like **iOS** and Android. Good rules of thumb are:

For world space positions and texture coordinates, use `float` precision.
For everything else (vectors, HDR colors, etc.), start with `half` precision. Increase only if necessary.
For very simple operations on texture data, use `fixed` precision.
In practice, exactly which number type you should use for depends on the platform and the GPU. Generally speaking:

All modern desktop GPUs will always compute everything in full `float` precision, so `float/half/fixed` end up being exactly the same underneath. This can make testing difficult, as it's harder to see if half/fixed precision is really enough, so always test your shaders on the target device for accurate results.
Mobile GPUs have actual `half` precision support. This is usually faster, and uses less power to do calculations.
`Fixed` precision is generally only useful for older mobile GPUs. Most modern GPUs (the ones that can run OpenGL ES 3 or Metal) internally treat `fixed` and `half` precision exactly the same.
See Data Types and Precision for more details.

Alpha Testing

The fixed-function AlphaTest - or its programmable equivalent, `clip()` - has different performance characteristics on different platforms:

Generally you gain a small advantage when using it to remove totally transparent pixels on most platforms. However, on PowerVR GPUs found in iOS and some Android devices, alpha testing is resource-intensive. Do not try to use it for performance optimization on these platforms, as it causes the game to run slower than usual.

Color Mask

On some platforms (mostly mobile GPUs found in iOS and Android devices), using ColorMask to leave out some channels (e.g. `ColorMask RGB`) can be resource-intensive, so only use it if really necessary.

Rendering with Replaced Shaders

Leave feedback

Some **rendering** effects require **rendering** a **scene** with a different set of shaders. For example, good edge detection would need a texture with **scene** normals, so it could detect edges where surface orientations differ. Other effects might need a texture with **scene** depth, and so on. To achieve this, it is possible to render the **scene** with replaced shaders of all objects.

Shader replacement is done from scripting using Camera.RenderWithShader or Camera.SetReplacementShader functions. Both functions take a **shader** and a **replacementTag**.

It works like this: the **camera** renders the scene as it normally would. the objects still use their materials, but the actual shader that ends up being used is changed:

If **replacementTag** is empty, then all objects in the scene are rendered with the given replacement shader.
If **replacementTag** is not empty, then for each object that would be rendered:
The real object's shader is queried for the tag value.
If it does not have that tag, object is **not rendered**.
A subshader is found in the replacement shader that has a given tag with the found value. If no such subshader is found, object is **not rendered**.
Now that subshader is used to render the object.
So if all shaders would have, for example, a "RenderType" tag with values like "Opaque", "Transparent", "Background", "Overlay", you could write a replacement shader that only renders solid objects by using one subshader with RenderType=Solid tag. The other tag types would not be found in the replacement shader, so the objects would be not rendered. Or you could write several subshaders for different "RenderType" tag values. Incidentally, all built-in Unity shaders have a "RenderType" tag set.

Lit shader replacement

When using shader replacement the scene is rendered using the render path that is configured on the camera. This means that the shader used for replacement can contain shadow and lighting passes (you can use surface shaders for shader replacement). This can be useful for doing rendering of special effects and scene debugging.

Shader replacement tags in built-in Unity shaders

All built-in Unity shaders have a "**RenderType**" tag set that can be used when rendering with replaced shaders. Tag values are the following:

Opaque: most of the shaders (Normal, Self Illuminated, Reflective, terrain shaders).
Transparent: most semitransparent shaders (Transparent, Particle, Font, terrain additive pass shaders).
TransparentCutout: masked transparency shaders (Transparent Cutout, two pass vegetation shaders).
Background: **Skybox** shaders.
Overlay: GUITexture, Halo, Flare shaders.
TreeOpaque: **terrain** engine tree bark.
TreeTransparentCutout: terrain engine tree leaves.
TreeBillboard: terrain engine billboarded trees.

Grass: terrain engine grass.
GrassBillboard: terrain engine billboarded grass.

Built-in scene depth/normals texture

A Camera has a built-in capability to render depth or depth+normals texture, if you need that in some of your effects. See Camera Depth Texture page. Note that in some cases (depending on the hardware), the depth and depth+normals textures can internally be rendered using shader replacement. So it is important to have the correct "**RenderType**" tag in your shaders.

Code Example

Your Start() function specifies the replacement shaders:

```
void Start() {
    camera.SetReplacementShader (EffectShader, "RenderType");
}
```

This requests that the EffectShader will use the RenderType key. The EffectShader will have key-value tags for each RenderType that you want. The Shader will look something like:

```
Shader "EffectShader" {
     SubShader {
         Tags { "RenderType"="Opaque" }
         Pass {
             ...
         }
     }
     SubShader {
         Tags { "RenderType"="SomethingElse" }
         Pass {
             ...
         }
     }
 ...
 }
```

SetReplacementShader will look through all the objects in the scene and, instead of using their normal shader, use the first subshader which has a matching value for the specified key. In this example, any objects whose shader has Rendertype="Opaque" tag will be replaced by first subshader in EffectShader, any objects with

RenderType="SomethingElse" shader will use second replacement subshader and so one. Any objects whose shader does not have a matching tag value for the specified key in the replacement shader will not be rendered.

Custom Shader GUI

Leave feedback

Sometimes you have a **shader** with some interesting data types that can not be nicely represented using the built in Unity material editor. Unity provides a way to override the default way **shader** properties are presented so that you can define your own. You can use this feature to define custom controls and data range validation.

The first part to writing custom editor for your shader's gui is defining a shader that requires a Custom Editor. The name you use for the custom editor is the class that will be looked up by Unity for the material editor.

To define a custom editor you extend from the ShaderGUI class and place the script below an Editor folder in the **assets** directory.

```
using UnityEditor;

public class CustomShaderGUI : ShaderGUI
{
    public override void OnGUI (MaterialEditor materialEditor, MaterialProperty[
    {
        base.OnGUI (materialEditor, properties);
    }
}
```

Any shader that has a custom editor defined (**CustomEditor "CustomShaderGUI"**) will instantiate an instance of the shader gui class listed above and execute the associated code.

A simple example

So we have a situation where we have a shader that can work in two modes; it renders standard diffuse lighting or it renders the blue and green channels with 50%.

```
Shader "Custom/Redify" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert addshadow
        #pragma shader_feature REDIFY_ON

        sampler2D _MainTex;
```

```
        struct Input {
            float2 uv_MainTex;
        };

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex);
            o.Albedo = c.rgb;
            o.Alpha = c.a;

            #if REDIFY_ON
            o.Albedo.gb *= 0.5;
            #endif
        }
        ENDCG
    }
    CustomEditor "CustomShaderGUI"
}
```

As you can see the shader has a Keyword available for setting: REDIFY_ON. This can be changed be set on a per material basis by using the shaderKeywords property of the material. Below is an ShaderGUI instance that does this.

```
using UnityEngine;
using UnityEditor;
using System;

public class CustomShaderGUI : ShaderGUI
{
    public override void OnGUI(MaterialEditor materialEditor, MaterialProperty[]
    {
        // render the default gui
        base.OnGUI(materialEditor, properties);

        Material targetMat = materialEditor.target as Material;

        // see if redify is set, and show a checkbox
        bool redify = Array.IndexOf(targetMat.shaderKeywords, "REDIFY_ON") != -1
        EditorGUI.BeginChangeCheck();
        redify = EditorGUILayout.Toggle("Redify material", redify);
        if (EditorGUI.EndChangeCheck())
        {
            // enable or disable the keyword based on checkbox
```

```
            if (redify)
                targetMat.EnableKeyword("REDIFY_ON");
            else
                targetMat.DisableKeyword("REDIFY_ON");
        }
    }
}
```

For a more comprehensive ShaderGUI example see the StandardShaderGUI.cs file together with the Standard.shader found in the 'Built-in shaders' package that can be downloaded from Unity Download Archive.

Note that the simple example above could also be solved much simpler using MaterialPropertyDrawers. Add the following line to the Properties section of the Custom/Redify shader:

```
[Toggle(REDIFY_ON)] _Redify("Red?", Int) = 0
```

and remove the:

```
CustomEditor "CustomShaderGUI"
```

Also see: MaterialPropertyDrawer

ShaderGUI should be used for more complex shader gui solutions where where e.g. material properties have dependencies on each other or special layout is wanted. You can combine using MaterialPropertyDrawers with ShaderGUI classes, see StandardShaderGUI.cs.

Using Depth Textures

Leave feedback

It is possible to create Render Textures where each **pixel** contains a high-precision Depth value. This is mostly used when some effects need the **Scene**'s Depth to be available (for example, soft particles, screen space ambient occlusion and translucency would all need the Scene's Depth). Image Effects often use Depth Textures too.

Pixel values in the Depth Texture range between 0 and 1, with a non-linear distribution. Precision is usually 32 or 16 bits, depending on configuration and platform used. When reading from the Depth Texture, a high precision value in a range between 0 and 1 is returned. If you need to get distance from the **Camera**, or an otherwise linear 0–1 value, compute that manually using helper macros (see below).

Depth Textures are supported on most modern hardware and graphics APIs. Special requirements are listed below:

Direct3D 11+ (Windows), OpenGL 3+ (Mac/Linux), OpenGL ES 3.0+ (Android/iOS), Metal (iOS) and consoles like **PS4**/**Xbox One** all support depth textures.
OpenGL ES 2.0 (iOS/Android) requires GL_OES_depth_texture extension to be present.
WebGL requires WEBGL_depth_texture extension.

Depth Texture Shader helper macros

Most of the time, Depth Texture are used to render Depth from the Camera. The UnityCG.cginc include file contains some macros to deal with the above complexity in this case:

UNITY_TRANSFER_DEPTH(o): computes eye space depth of the vertex and outputs it in **o** (which must be a float2). Use it in a vertex program when **rendering** into a depth texture. On platforms with native depth textures this macro does nothing at all, because Z buffer value is rendered implicitly.
UNITY_OUTPUT_DEPTH(i): returns eye space depth from **i** (which must be a float2). Use it in a fragment program when rendering into a depth texture. On platforms with native depth textures this macro always returns zero, because Z buffer value is rendered implicitly.
COMPUTE_EYEDEPTH(i): computes eye space depth of the vertex and outputs it in **o**. Use it in a vertex program when **not** rendering into a depth texture.
DECODE_EYEDEPTH(i)/LinearEyeDepth(i): given high precision value from depth texture **i**, returns corresponding eye space depth.
Linear01Depth(i): given high precision value from depth texture **i**, returns corresponding linear depth in range between 0 and 1.
Note: On DX11/12, PS4, XboxOne and Metal, the Z buffer range is 1–0 and UNITY_REVERSED_Z is defined. On other platforms, the range is 0–1.

For example, this **shader** would render depth of its **GameObjects**:

```
Shader "Render Depth" {
    SubShader {
        Tags { "RenderType"="Opaque" }
        Pass {
            CGPROGRAM
```

```
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct v2f {
                float4 pos : SV_POSITION;
                float2 depth : TEXCOORD0;
            };

            v2f vert (appdata_base v) {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                UNITY_TRANSFER_DEPTH(o.depth);
                return o;
            }

            half4 frag(v2f i) : SV_Target {
                UNITY_OUTPUT_DEPTH(i.depth);
            }
            ENDCG
        }
    }
}
```

See also

Camera Depth Textures
Writing Image Effects

Camera's Depth Texture

Leave feedback

A Camera can generate a depth, depth+normals, or motion vector Texture. This is a minimalistic G-buffer Texture that can be used for post-processing effects or to implement custom lighting models (e.g. light pre-pass). It is also possible to build similar textures yourself, using Shader Replacement feature.

The Camera's depth Texture mode can be enabled using Camera.depthTextureMode variable from script.

There are three possible depth texture modes:

DepthTextureMode.Depth: a depth texture.
DepthTextureMode.DepthNormals: depth and view space normals packed into one texture.*
DepthTextureMode.MotionVectors: per-pixel screen space motion of each screen texel for the current frame. Packed into a RG16 texture.
These are flags, so it is possible to specify any combination of the above textures.

DepthTextureMode.Depth texture

This builds a screen-sized depth texture.

Depth texture is rendered using the same **shader** passes as used for shadow caster **rendering** (`ShadowCaster` pass type). So by extension, if a shader does not support shadow casting (i.e. there's no shadow caster pass in the shader or any of the fallbacks), then objects using that shader will not show up in the depth texture.

Make your shader fallback to some other shader that has a shadow casting pass, or
If you're using surface shaders, adding an `addshadow` directive will make them generate a shadow pass too.
Note that only "opaque" objects (that which have their materials and shaders setup to use render queue <= 2500) are rendered into the depth texture.

DepthTextureMode.DepthNormals texture

This builds a screen-sized 32 bit (8 bit/channel) texture, where view space normals are encoded into R&G channels, and depth is encoded in B&A channels. Normals are encoded using Stereographic projection, and depth is 16 bit value packed into two 8 bit channels.

`UnityCG.cginc` include file has a helper function `DecodeDepthNormal` to decode depth and normal from the encoded **pixel** value. Returned depth is in 0..1 range.

For examples on how to use the depth and normals texture, please refer to the EdgeDetection image effect in the Shader Replacement example project or Screen Space Ambient Occlusion Image Effect.

DepthTextureMode.MotionVectors texture

This builds a screen-sized RG16 (16-bit float/channel) texture, where screen space pixel motion is encoded into the R&G channels. The pixel motion is encoded in screen UV space.

When sampling from this texture motion from the encoded pixel is returned in a rance of –1..1. This will be the UV offset from the last frame to the current frame.

Tips & Tricks

Camera inspector indicates when a camera is rendering a depth or a depth+normals texture.

The way that depth textures are requested from the Camera (Camera.depthTextureMode) might mean that after you disable an effect that needed them, the Camera might still continue rendering them. If there are multiple effects present on a Camera, where each of them needs the depth texture, there's no good way to automatically disable depth texture rendering if you disable the individual effects.

When implementing complex Shaders or Image Effects, keep Rendering Differences Between Platforms in mind. In particular, using depth texture in an Image Effect often needs special handling on Direct3D + Anti-Aliasing.

In some cases, the depth texture might come directly from the native Z buffer. If you see artifacts in your depth texture, make sure that the shaders that use it **do not** write into the Z buffer (use ZWrite Off).

Shader variables

Depth textures are available for sampling in shaders as global shader properties. By declaring a sampler called `_CameraDepthTexture` you will be able to sample the main depth texture for the camera.

`_CameraDepthTexture` always refers to the camera's primary depth texture. By contrast, you can use `_LastCameraDepthTexture` to refer to the last depth texture rendered by any camera. This could be useful for example if you render a half-resolution depth texture in script using a secondary camera and want to make it available to a post-process shader.

The motion vectors texture (when enabled) is available in Shaders as a global Shader property. By declaring a sampler called '_CameraMotionVectorsTexture' you can sample the Texture for the curently rendering Camera.

Under the hood

Depth textures can come directly from the actual depth buffer, or be rendered in a separate pass, depending on the rendering path used and the hardware. Typically when using Deferred Shading or Legacy Deferred Lighting rendering paths, the depth textures come "for free" since they are a product of the G-buffer rendering anyway.

When the DepthNormals texture is rendered in a separate pass, this is done through Shader Replacement. Hence it is important to have correct "**RenderType**" tag in your shaders.

When enabled, the MotionVectors texture always comes from a extra render pass. Unity will render moving GameObjects into this buffer, and construct their motion from the last frame to the current frame.

See also

Writing Image Effects
Depth Textures
Shader Replacement

Platform-specific rendering differences

Leave feedback

Unity runs on various graphics library platforms: Open GL, Direct3D, Metal, and games consoles. In some cases, there are differences in how graphics **rendering** behaves between the platforms and **Shader** language semantics. Most of the time the Unity Editor hides the differences, but there are some situations where the Editor cannot do this for you. In these situations, you need to ensure that you negate differences between the platforms. These situations, and the actions you need to take if they occur, are listed below.

Render Texture coordinates

Vertical Texture coordinate conventions differ between two types of platforms: Direct3D-like and OpenGL-like.

Direct3D-like: The coordinate is 0 at the top and increases downward. This applies to Direct3D, Metal and consoles.
OpenGL-like: The coordinate is 0 at the bottom and increases upward. This applies to OpenGL and OpenGL ES.
This difference tends not to have any effect on your project, other than when rendering into a Render Texture. When rendering into a Texture on a Direct3D-like platform, Unity internally flips rendering upside down. This makes the conventions match between platforms, with the OpenGL-like platform convention the standard.

Image Effects and rendering in UV space are two common cases in the Shaders where you need to take action to ensure that the different coordinate conventions do not create problems in your project.

Image Effects

When you use Image Effects and anti-aliasing, the resulting source Texture for an Image Effect is not flipped to match the OpenGL-like platform convention. In this case, Unity renders to the screen to get anti-aliasing and then resolves rendering into a Render Texture for further processing with an Image Effect.

If your Image Effect is a simple one that processes one Render Texture at a time, Graphics.Blit deals with the inconsistent coordinates. However, if you're processing more than one Render Texture together in your Image Effect, the Render Textures are likely to come out at different vertical orientations in Direct3D-like platforms and when you use anti-aliasing. To standardise the coordinates, you need to manually "flip" the screen Texture upside down in your **Vertex Shader** so that it matches the OpenGL-like coordinate standard.

The following code sample demonstrates how to do this:

```
// Flip sampling of the Texture:
// The main Texture
// texel size will have negative Y).

#if UNITY_UV_STARTS_AT_TOP
if (_MainTex_TexelSize.y < 0)
        uv.y = 1-uv.y;
#endif
```

Refer to the Edge Detection **Scene** in Unity's Shader Replacement sample project (see Unity's Learn resources) for a more detailed example of this. Edge detection in this project uses both the screen Texture and the **Camera**'s Depth+Normals texture.

A similar situation occurs with GrabPass. The resulting render Texture might not actually be turned upside down on Direct3D-like (non-OpenGL-like) platforms. If your Shader code samples GrabPass Textures, use the `ComputeGrabScreenPos` function from the UnityCG include file.

Rendering in UV space

When rendering in Texture coordinate (UV) space for special effects or tools, you might need to adjust your Shaders so that rendering is consistent between Direct3D-like and OpenGL-like systems. You also might need to adjust your rendering between rendering into the screen and rendering into a Texture. Adjust these by flipping the Direct3D-like projection upside down so its coordinates match the OpenGL-like projection coordinates.

The built-in variable `ProjectionParams.x` contains a `+1` or `-1` value. `-1` indicates a projection has been flipped upside down to match OpenGL-like projection coordinates, while `+1` indicates it hasn't been flipped. You can check this value in your Shaders and then perform different actions. The example below checks if a projection has been flipped and, if so, flips and then returns the UV coordinates to match.

```
float4 vert(float2 uv : TEXCOORD0) : SV_POSITION
{
    float4 pos;
    pos.xy = uv;
    // This example is rendering with upside-down flipped projection,
    // so flip the vertical UV coordinate too
    if (_ProjectionParams.x < 0)
        pos.y = 1 - pos.y;
    pos.z = 0;
    pos.w = 1;
    return pos;
}
```

Clip space coordinates

Similar to Texture coordinates, the clip space coordinates (also known as post-projection space coordinates) differ between Direct3D-like and OpenGL-like platforms:

Direct3D-like: The clip space depth goes from 0.0 at the near plane to +1.0 at the far plane. This applies to Direct3D, Metal and consoles.

OpenGL-like: The clip space depth goes from –1.0 at the near plane to +1.0 at the far plane. This applies to OpenGL and OpenGL ES.

Inside Shader code, you can use the `UNITY_NEAR_CLIP_VALUE` built-in macro to get the near plane value based on the platform.

Inside script code, use GL.GetGPUProjectionMatrix to convert from Unity's coordinate system (which follows OpenGL-like conventions) to Direct3D-like coordinates if that is what the platform expects.

Precision of Shader computations

To avoid precision issues, make sure that you test your Shaders on the target platforms. The GPUs in mobile devices and PCs differ in how they treat floating point types. PC GPUs treat all floating point types (float, half and fixed) as the same - they do all calculations using full 32-bit precision, while many mobile device GPUs do not do this.

See documentation on data types and precision for details.

Const declarations in Shaders

Use of `const` differs between Microsoft HSL (see msdn.microsoft.com) and OpenGL's GLSL (see Wikipedia) Shader language.

Microsoft's HLSL `const` has much the same meaning as it does in C# and C++ in that the variable declared is read-only within its scope but can be initialized in any way.

OpenGL's GLSL `const` means that the variable is effectively a compile time constant, and so it must be initialized with compile time constraints (either literal values or calculations on other `const`s).

It is best to follow the OpenGL's GLSL semantics and only declare a variable as `const` when it is truly invariant. Avoid initializing a `const` variable with some other mutable values (for example, as a local variable in a function). This also works in Microsoft's HLSL, so using `const` in this way avoids confusing errors on some platforms.

Semantics used by Shaders

To get Shaders working on all platforms, some Shader values should use these semantics:

Vertex Shader output (clip space) position: `SV_POSITION`. Sometimes Shaders use POSITION semantics to get Shaders working on all platforms. Note that this does not not work on Sony **PS4** or with tessellation.

Fragment Shader output color: `SV_Target`. Sometimes Shaders use `COLOR` or `COLOR0` to get Shaders working on all platforms. Note that this does not work on Sony PS4.

When rendering Meshes as Points, output `PSIZE` semantics from the vertex Shader (for example, set it to 1). Some platforms, such as OpenGL ES or Metal, treat point size as "undefined" when it's not written to from the Shader.

See documentation on Shader semantics for more details.

Direct3D Shader compiler syntax

Direct3D platforms use Microsoft's HLSL Shader compiler. The HLSL compiler is stricter than other compilers about various subtle Shader errors. For example, it doesn't accept function output values that aren't initialized properly.

The most common situations that you might run into using this are:

A Surface Shader vertex modifier that has an `out` parameter. Initialize the output like this:

```
void vert (inout appdata_full v, out Input o)
    {
      **UNITY_INITIALIZE_OUTPUT(Input,o);**
      // ...
    }
```

Partially initialized values. For example, a function returns `float4` but the code only sets the `.xyz` values of it. Set all values or change to `float3` if you only need three values.

Using `tex2D` in the Vertex Shader. This is not valid, because UV derivatives don't exist in the vertex Shader. You need to sample an explicit mip level instead; for example, use `tex2Dlod` (`tex, float4(uv,0,0)`). You also need to add `#pragma target 3.0` as `tex2Dlod` is a Shader model 3.0 feature.

DirectX 11 (DX11) HLSL syntax in Shaders

Some parts of the Surface Shader compilation pipeline do not understand DirectX 11-specific HLSL (Microsoft's shader language) syntax.

If you're using HLSL features like `StructuredBuffers`, `RWTextures` and other non-DirectX 9 syntax, wrap them in a DirectX X11-only preprocessor macro as shown in the example below.

```
#ifdef SHADER_API_D3D11
// DirectX11-specific code, for example
StructuredBuffer<float4> myColors;
RWTexture2D<float4> myRandomWriteTexture;
#endif
```

Using Shader framebuffer fetch

Some GPUs (most notably PowerVR-based ones on iOS) allow you to do a form of programmable blending by providing current fragment color as input to the **Fragment Shader** (see `EXT_shader_framebuffer_fetch` on khronos.org).

It is possible to write Shaders in Unity that use the framebuffer fetch functionality. To do this, use the `inout` color argument when you write a Fragment Shader in either HLSL (Microsoft's shading language - see msdn.microsoft.com) or Cg (the shading language by Nvidia - see nvidia.co.uk).

The example below is in Cg.

```
CGPROGRAM
// only compile Shader for platforms that can potentially
// do it (currently gles,gles3,metal)
#pragma only_renderers framebufferfetch

void frag (v2f i, inout half4 ocol : SV_Target)
{
    // ocol can be read (current framebuffer color)
    // and written into (will change color to that one)
    // ...
}
ENDCG
```

The Depth (Z) direction in Shaders

Depth (Z) direction differs on different Shader platforms.

DirectX 11, DirectX 12, PS4, Xbox One, Metal: Reversed direction

The depth (Z) buffer is 1.0 at the near plane, decreasing to 0.0 at the far plane.

Clip space range is [near,0] (meaning the near plane distance at the near plane, decreasing to 0.0 at the far plane).

Other platforms: Traditional direction

The depth (Z) buffer value is 0.0 at the near plane and 1.0 at the far plane.

Clip space depends on the specific platform:

On Direct3D-like platforms, the range is [0,far] (meaning 0.0 at the near plane, increasing to the far plane distance at the far plane).
On OpenGL-like platforms, the range is [-near,far] (meaning minus the near plane distance at the near plane, increasing to the far plane distance at the far plane).
Note that reversed direction depth (Z), combined with a floating point depth buffer, significantly improves depth buffer precision against the traditional direction. The advantages of this are less conflict for Z coordinates and better shadows, especially when using small near planes and large far planes.

So, when you use Shaders from platforms with the depth (Z) reversed:

UNITY_REVERSED_Z is defined.
`_CameraDepth` Texture texture range is 1 (near) to 0 (far).
Clip space range is within "near" (near) to 0 (far).
However, the following macros and functions automatically work out any differences in depth (Z) directions:

`Linear01Depth(float z)`
`LinearEyeDepth(float z)`
UNITY_CALC_FOG_FACTOR(coord)

Fetching the depth Buffer

If you are fetching the depth (Z) buffer value manually, you might want to check the buffer direction. The following is an example of this:

```
float z = tex2D(_CameraDepthTexture, uv);
#if defined(UNITY_REVERSED_Z)
    z = 1.0f - z;
#endif
```

Using clip space

If you are using clip space (Z) depth manually, you might also want to abstract platform differences by using the following macro:

```
float clipSpaceRange01 = UNITY_Z_0_FAR_FROM_CLIPSPACE(rawClipSpace);
```

Note: This macro does not alter clip space on OpenGL or OpenGL ES platforms, so it returns within "-near"1 (near) to far (far) on these platforms.

Projection matrices

GL.GetGPUProjectionMatrix() returns a z-reverted matrix if you are on a platform where the depth (Z) is reversed. However, if you're composing from projection matrices manually (for example, for custom shadows or depth rendering), you need to revert depth (Z) direction yourself where it applies via script.

An example of this is below:

```
var shadowProjection = Matrix4x4.Ortho(...); //shadow camera projection matrix
var shadowViewMat = ...     //shadow camera view matrix
var shadowSpaceMatrix = ... //from clip to shadowMap texture space

//'m_shadowCamera.projectionMatrix' is implicitly reversed
//when the engine calculates device projection matrix from the camera projection
m_shadowCamera.projectionMatrix = shadowProjection;
```

```
//'shadowProjection' is manually flipped before being concatenated to 'm_shadowMatrix'
//because it is seen as any other matrix to a Shader.
if(SystemInfo.usesReversedZBuffer)
{
    shadowProjection[2, 0] = -shadowProjection[2, 0];
    shadowProjection[2, 1] = -shadowProjection[2, 1];
    shadowProjection[2, 2] = -shadowProjection[2, 2];
    shadowProjection[2, 3] = -shadowProjection[2, 3];
}
    m_shadowMatrix = shadowSpaceMatrix * shadowProjection * shadowViewMat;
```

Depth (Z) bias

Unity automatically deals with depth (Z) bias to ensure it matches Unity's depth (Z) direction. However, if you are using a native code rendering plugin, you need to negate (reverse) depth (Z) bias in your C or C++ code.

Tools to check for depth (Z) direction

Use SystemInfo.usesReversedZBuffer to find out if you are on a platform using reversed depth (Z).

Shader Level of Detail

Leave feedback

Shader Level of Detail (LOD) works by only using **shaders** or **subshaders** that have their LOD value less than a given number.

By default, allowed LOD level is infinite, that is, all shaders that are supported by the user's hardware can be used. However, in some cases you might want to drop shader details, even if the hardware can support them. For example, some cheap graphics cards might support all the features, but are too slow to use them. So you may want to not use parallax normal mapping on them.

Shader LOD can be either set per individual shader (using Shader.maximumLOD), or globally for all shaders (using Shader.globalMaximumLOD).

In your custom shaders, use **LOD** command to set up LOD value for any subshader.

Built-in shaders in Unity have their LODs set up this way:

VertexLit kind of shaders = 100
Decal, Reflective VertexLit = 150
Diffuse = 200
Diffuse Detail, Reflective Bumped Unlit, Reflective Bumped VertexLit = 250
Bumped, Specular = 300
Bumped Specular = 400
Parallax = 500
Parallax Specular = 600

Texture arrays

Leave feedback

Similar to regular 2D textures (Texture2D class, **sampler2D** in shaders), cube maps (Cubemap class, **samplerCUBE** in shaders), and 3D textures (Texture3D class, **sampler3D** in shaders), Unity also supports 2D texture arrays.

A texture array is a collection of same size/format/flags 2D textures that look like a single object to the GPU, and can be sampled in the **shader** with a texture element index. They are useful for implementing custom **terrain rendering** systems or other special effects where you need an efficient way of accessing many textures of the same size and format. Elements of a 2D texture array are also known as slices, or layers.

Platform Support

Texture arrays need to be supported by the underlying graphics API and the GPU. They are available on:

Direct3D 11/12 (Windows, Xbox One)
OpenGL Core (Mac OS X, Linux)
Metal (iOS, Mac OS X)
OpenGL ES 3.0 (Android, iOS, WebGL 2.0)
PlayStation 4
Other platforms do not support texture arrays (OpenGL ES 2.0 or WebGL 1.0). Use SystemInfo.supports2DArrayTextures to determine texture array support at runtime.

Creating and manipulating texture arrays

As there is no texture import pipeline for texture arrays, they must be created from within your **scripts**. Use the Texture2DArray class to create and manipulate them. Note that texture arrays can be serialized as **assets**, so it is possible to create and fill them with data from editor scripts.

Normally, texture arrays are used purely within GPU memory, but you can use Graphics.CopyTexture, Texture2DArray.GetPixels and Texture2DArray.SetPixels to transfer **pixels** to and from system memory.

Using texture arrays as render targets

Texture array elements may also be used as render targets. Use RenderTexture.dimension to specify in advance whether the render target is to be a 2D texture array. The **depthSlice** argument to Graphics.SetRenderTarget specifies which mipmap level or cube map face to render to. On platforms that support "layered rendering" (i.e. geometry shaders), you can set the **depthSlice** argument to –1 to set the whole texture array as a render target. You can also use a geometry shader to render into individual elements.

Using texture arrays in shaders

Since texture arrays do not work on all platforms, shaders need to use an appropriate compilation target or feature requirement to access them. The minimum shader model compilation target that supports texture arrays is `3.5`, and the feature name is `2darray`.

Use these macros to declare and sample texture arrays:

UNITY_DECLARE_TEX2DARRAY(name) declares a texture array sampler variable inside HLSL code.
UNITY_SAMPLE_TEX2DARRAY(name,uv) samples a texture array with a **float3** UV; the z **component** of the coordinate is an array element index.
UNITY_SAMPLE_TEX2DARRAY_LOD(name,uv,lod) samples a texture array with an explicit mipmap level.

Examples

The following shader example samples a texture array using object space vertex positions as coordinates:

```
Shader "Example/Sample2DArrayTexture"
{
    Properties
```

```
    {
        _MyArr ("Tex", 2DArray) = "" {}
        _SliceRange ("Slices", Range(0,16)) = 6
        _UVScale ("UVScale", Float) = 1.0
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // texture arrays are not available everywhere,
            // only compile shader on platforms where they are
            #pragma require 2darray

            #include "UnityCG.cginc"

            struct v2f
            {
                float3 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
            };

            float _SliceRange;
            float _UVScale;

            v2f vert (float4 vertex : POSITION)
            {
                v2f o;
                o.vertex = mul(UNITY_MATRIX_MVP, vertex);
                o.uv.xy = (vertex.xy + 0.5) * _UVScale;
                o.uv.z = (vertex.z + 0.5) * _SliceRange;
                return o;
            }

            UNITY_DECLARE_TEX2DARRAY(_MyArr);

            half4 frag (v2f i) : SV_Target
            {
                return UNITY_SAMPLE_TEX2DARRAY(_MyArr, i.uv);
            }
            ENDCG
        }
    }
}
```

See Also

Introduction To Textures in Direct3D documentation.
Array Textures in OpenGL Wiki.

2018–03–20 Page amended with editorial review

Shader #pragma directives added in Unity 2018.1

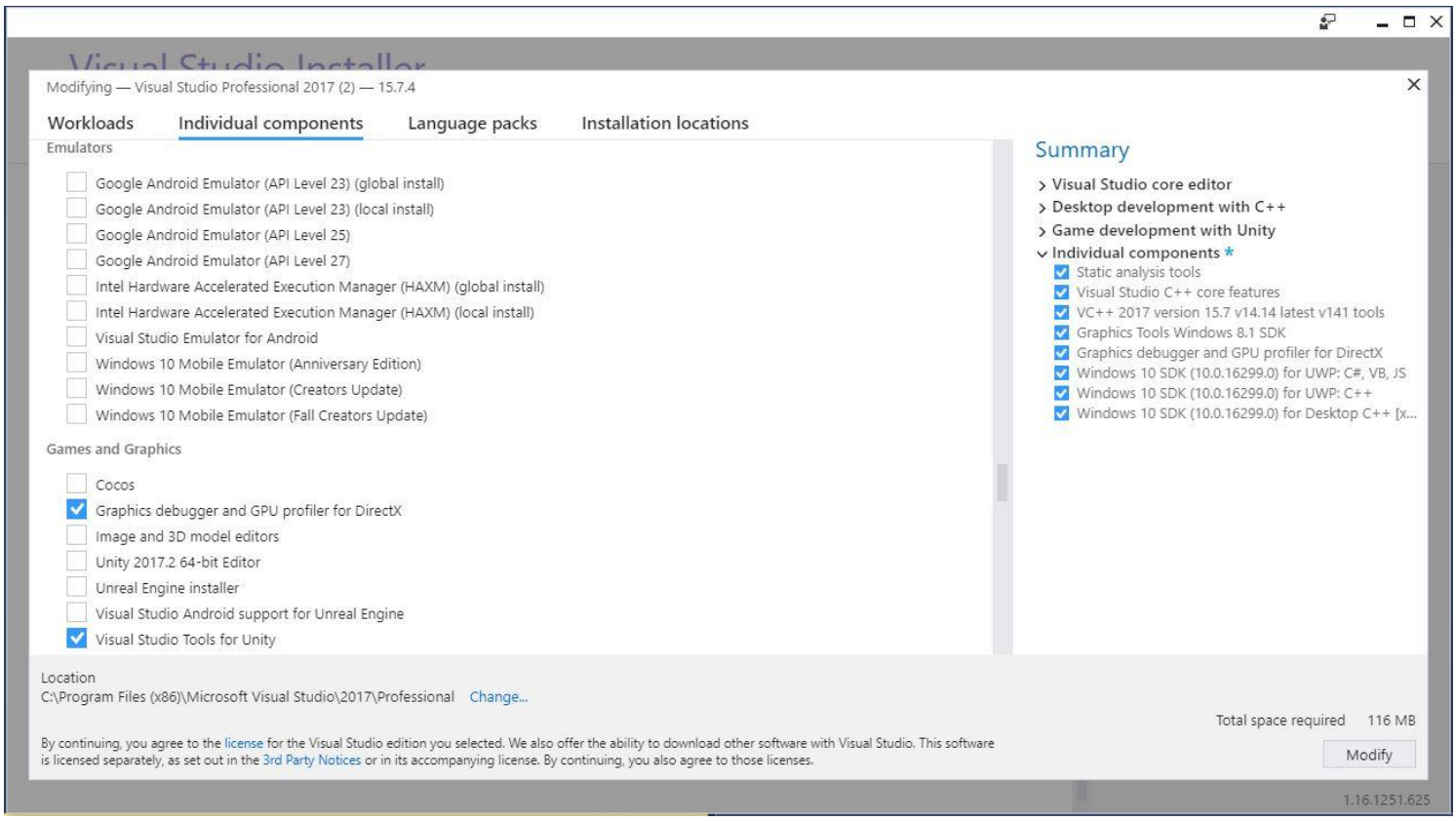
Debugging DirectX 11/12 shaders with Visual Studio

Leave feedback

Use the Graphics Debugger in Microsoft Visual Studio (2012 version or later) to capture individual frames of applications for debugging purposes, from platforms like Unity Editor, Windows Standalone or **Universal Windows Platform**.

To install the Graphics Debugger in Visual Studio:

Go to **Tools** > **Get Tools and Features**



On the **Individual components** tab, scroll to **Games and Graphics** and check the box for **Graphics debugger and GPU** profiler__ for DirectX__

Click **Modify**

Wait for installation, then follow the instructions to restart your computer

Capture DirectX shaders with Visual Studio

You should use a built version of your Unity application to capture frames, rather than a version running in the Unity Editor. This is because the Editor might have multiple child windows open at once, and the Graphics Debugger might capture a frame from an unintended window.

Steps to capture a frame from Unity Editor or Windows Standalone

To use the Graphics Debugger on either of these two platforms, you need to create a dummy Visual Studio Project:

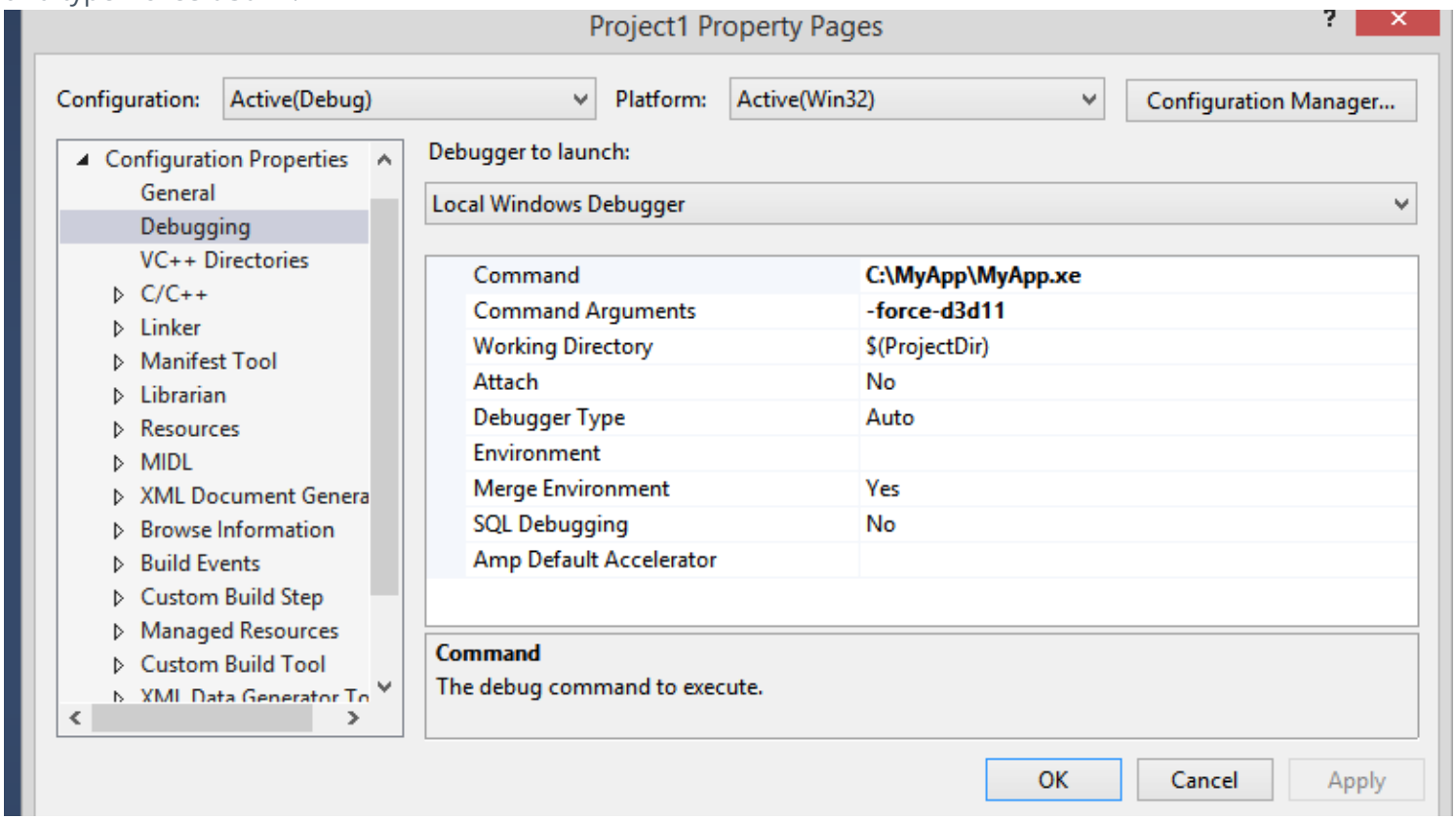
Launch Visual Studio 2017

Go to **File** > **New** > **Project** > **Visual C++** > **Empty Project**

Go to **Project** > **Properties** > **Configuration Properties** > **Debugging**

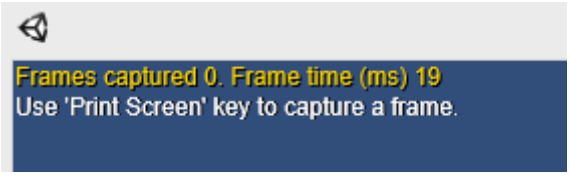
In the **Command** field, replace $(TargetPath) with the path to the Unity Editor or Windows Standalone (for example, *C:\MyApp\MyApp.exe*)

If you want to force Windows Standalone or Unity Editor to run under DirectX 11, select **Command Arguments** and type **-force-d3d11**.

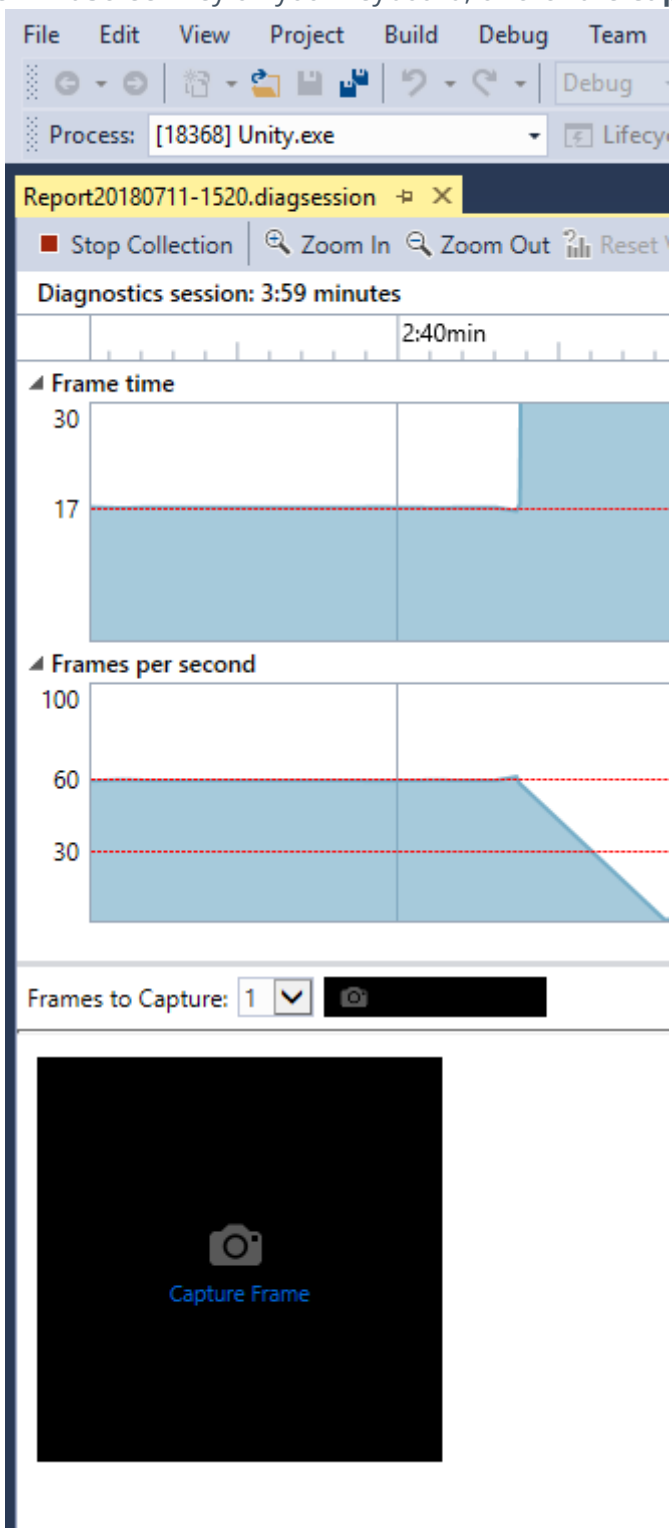


Go to **Debug** > **Graphics** > **Start Graphics Debugging**

If everything is configured correctly, Unity displays the following text in the top-left corner of the application:



To capture a frame, use the Print Screen key on your keyboard, or click the **Capture Frame** box on the left side of



the Visual Studio interface.

Debug DirectX shaders with Visual Studio

To debug a shader, you have to compile with debug symbols. To do that, you need to insert **#pragma enable_d3d11_debug_symbols.**

Your shader should look something like this:

```
Shader "Custom/NewShader" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert
        #pragma enable_d3d11_debug_symbols
        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex);
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Example workflow

Let's create a basic example to show the entire process:

Create a new Unity project (see documentation on Getting Started).

In the top menu, go to **Assets** > **Create** > **Shader** > **Standard Surface Shader**. This creates a new shader file in your **Project** folder.

Select the shader file, and in the **Inspector** window, click **Open**. This opens the shader file in your scripting editor. Insert `#pragma enable_d3d11_debug_symbols` into the shader code, underneath the other `#pragma` lines.

Create a new Material (menu: **Assets** > **Create** > **Material**).

In the Material's Inspector window, select the **Shader** dropdown, go to **Custom**, and select the shader you just created.

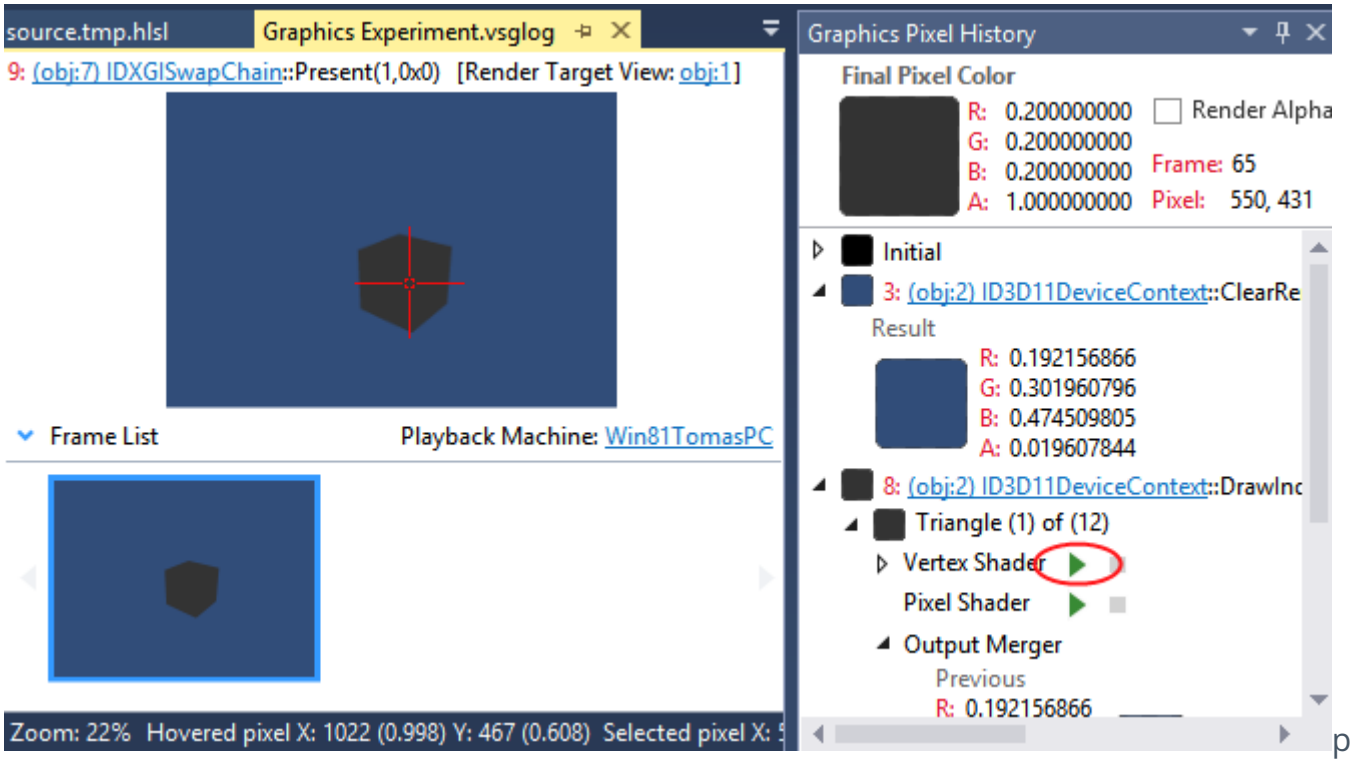
Create a 3D cube GameObject (menu: **GameObject** > **3D Object** > **Cube**).

Assign your new Material to your new GameObject. To do this, drag the Material from the **Project** window to the 3D cube.

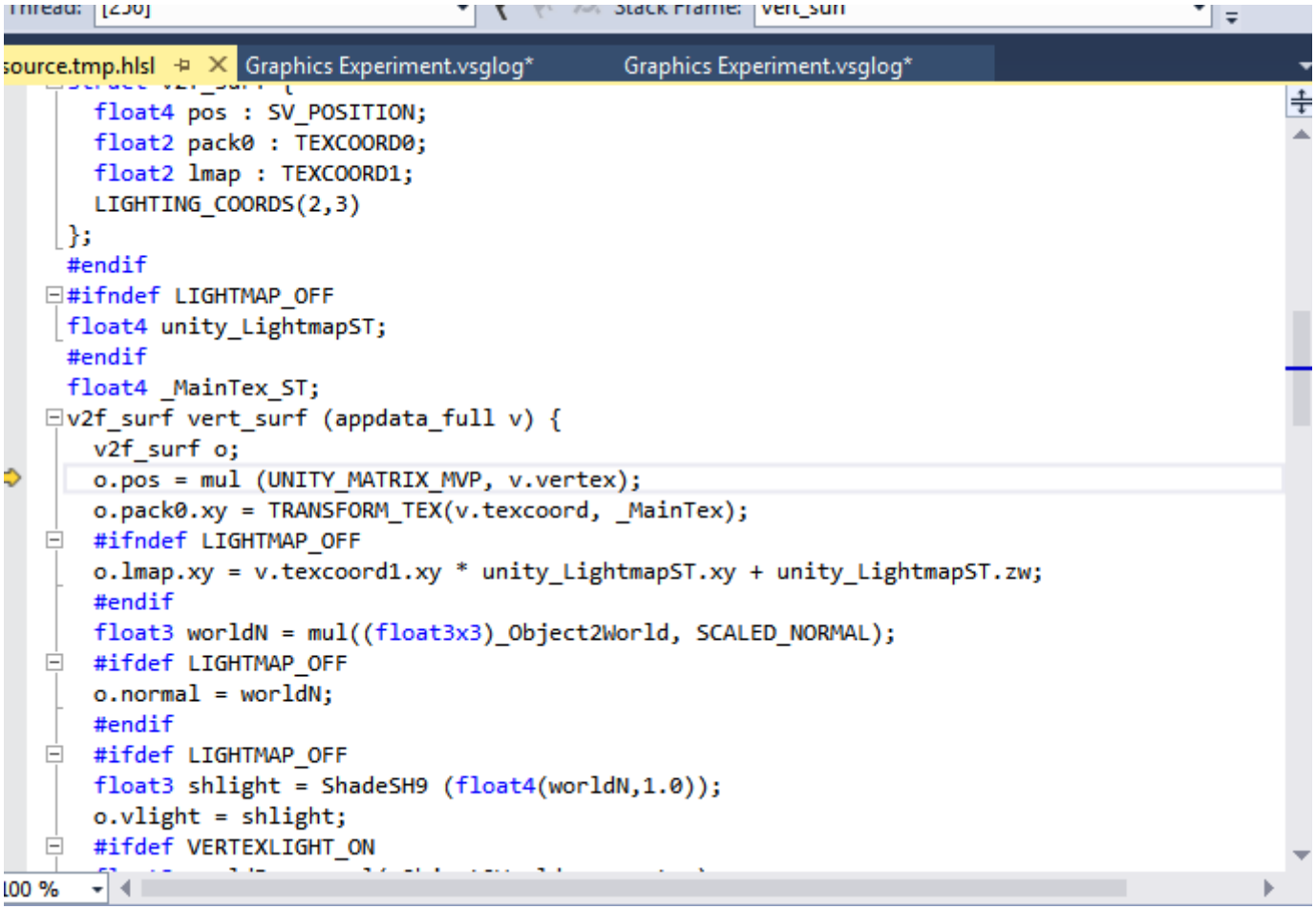
Build the project for Windows Standalone. Note that real projects might be so large that building them every time you want to debug a shader becomes inefficient; in that case, debug in the Editor, but make sure your capture has profiled the correct window.

Capture a frame, using the steps described above in the section **Capture DirectX shaders with Visual Studio**.

Your captured frame appears in Visual Studio. Right-click it, and select **Pixel.**



pixelThe smallest unit in a computer image. Pixel size depends on your screen resolution. Pixel lighting is calculated at every screen pixel. More infoSee in Glossary of an object which has the custom shader assigned.">
Captured frame, **History**, and selecting a **pixel** of an object which has the custom shader assigned.
Click the play button next to the **Vertex Shader** (highlighted in the screenshot above). This opens the vertex shader file:



There is a known issues while working with DirectX 12, in which the play buttons are not available, and the following error appears: This draw call uses system-value semantics that interfere with pixel history computation. If you experience this, use PIX to debug your shaders instead.

Universal Windows Platform

When you debug for Universal Windows Platform, you don't need to create a dummy Visual Studio project, because Unity creates it for you.

Steps to capture a frame and begin shader debugging are the same as they are for the Unity Editor or Windows Standalone.

Alternative shader debugging techniques

You can also use RenderDoc to debug shaders. In RenderDoc, you capture the **Scene** from within the Editor, then use the standalone tool for debugging.

PIX works in a similar way to Visual Studio's Graphics Debugger. Use PIX instead of the Graphics Debugger to debug DirectX 12 projects.

2018–09–14 Page published with editorial review

Debugging DirectX 12 shaders with PIX

Leave feedback

PIX is a performance tuning and debugging tool by Microsoft, for Windows developers. It offers a range of modes for analyzing an application's performance, and includes the ability to capture frames of DirectX projects from an application for debugging.

Use PIX to investigate issues in Windows 64-bit (x86_64) Standalone or **Universal Windows Platform** applications.

To install PIX, download and run the Microsoft PIX installer and follow the instructions.

For more information about PIX, see Microsoft's PIX Introduction and PIX Documentation.

Debugging DirectX shaders with PIX

You should use a built version of your Unity application to capture frames, rather than a version running in the Unity Editor. This is because you need to launch the target application from within PIX to capture GPU frames.

Using a development build adds additional information to PIX, which makes navigating the **scene** capture easier.

Create a project with a debug-enabled Shader

To debug the shader with source code in PIX, you need to insert the following pragma into the shader code: `#pragma enable_d3d11_debug_symbols`

Example

The following walkthrough uses a basic example to demonstrate the entire process.

Create a basic project:

Create a new Unity project (see documentation on Getting Started).

In the top menu, go to **Assets** > **Create** > **Shader** > **Standard Surface Shader**. This creates a new shader file in your **Project** folder.

Select the shader file, and in the **Inspector** window, click **Open**. This opens the shader file in your scripting editor. Insert `#pragma enable_d3d11_debug_symbols` into the shader code, underneath the other `#pragma` lines.

Create a new Material (menu: **Assets** > **Create** > **Material**).

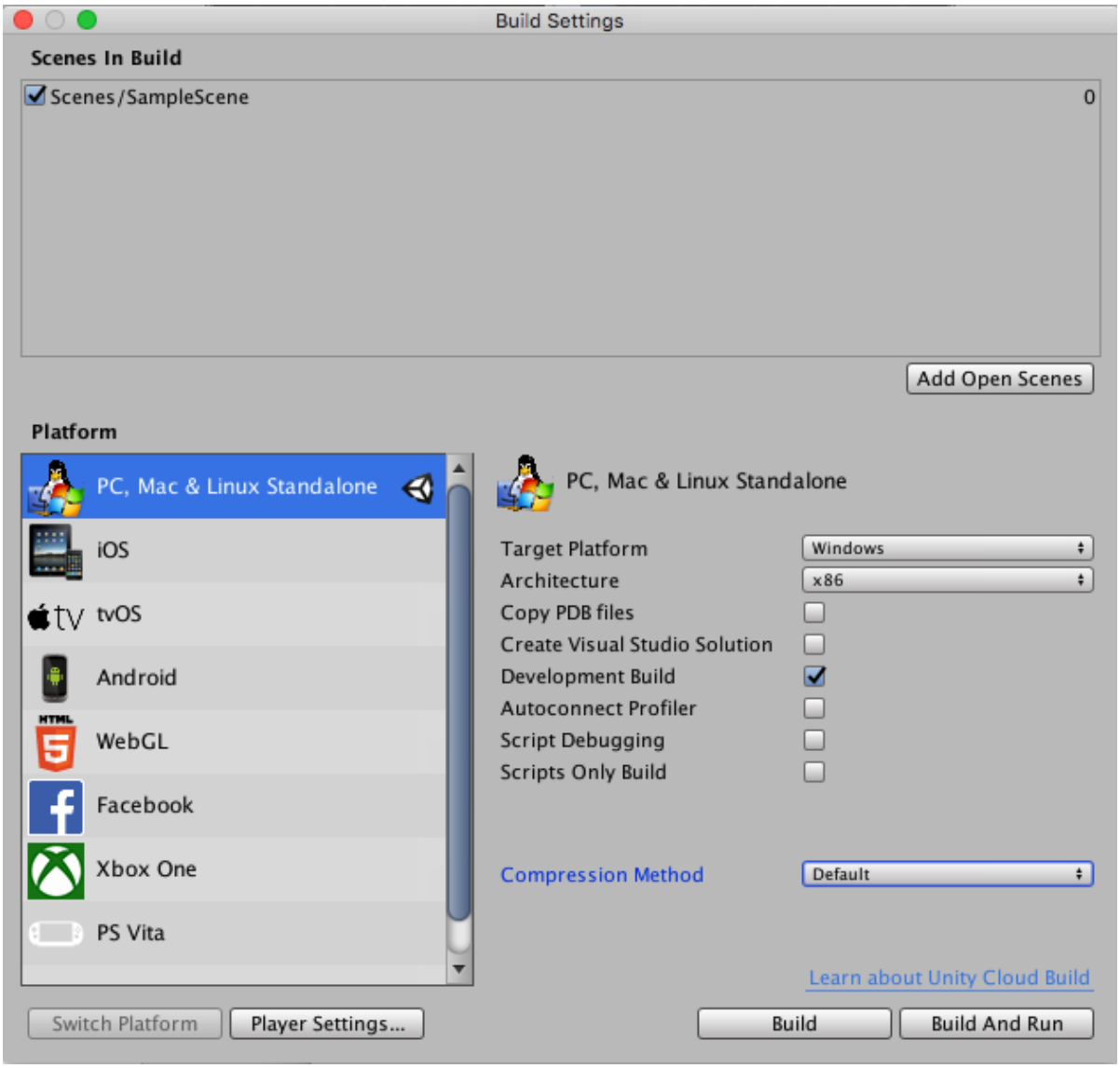
In the Material's Inspector window, select the **Shader** dropdown, go to **Custom**, and select the shader you just created.

Create a 3D cube GameObject (menu: **GameObject** > **3D Object** > **Cube**).

Assign your new Material to your new GameObject. To do this, drag the Material from the **Project** window to the 3D cube.

Capture a frame from a Windows Standalone application:

Go to **File** > **Build Settings**, and under **Platform**, select **PC, Mac & Linux Standalone**. Set the **Target Platform** to **Windows**, set the **Architecture** to **x86_64**, and click the **Development Build** checkbox.



Click **Build**.

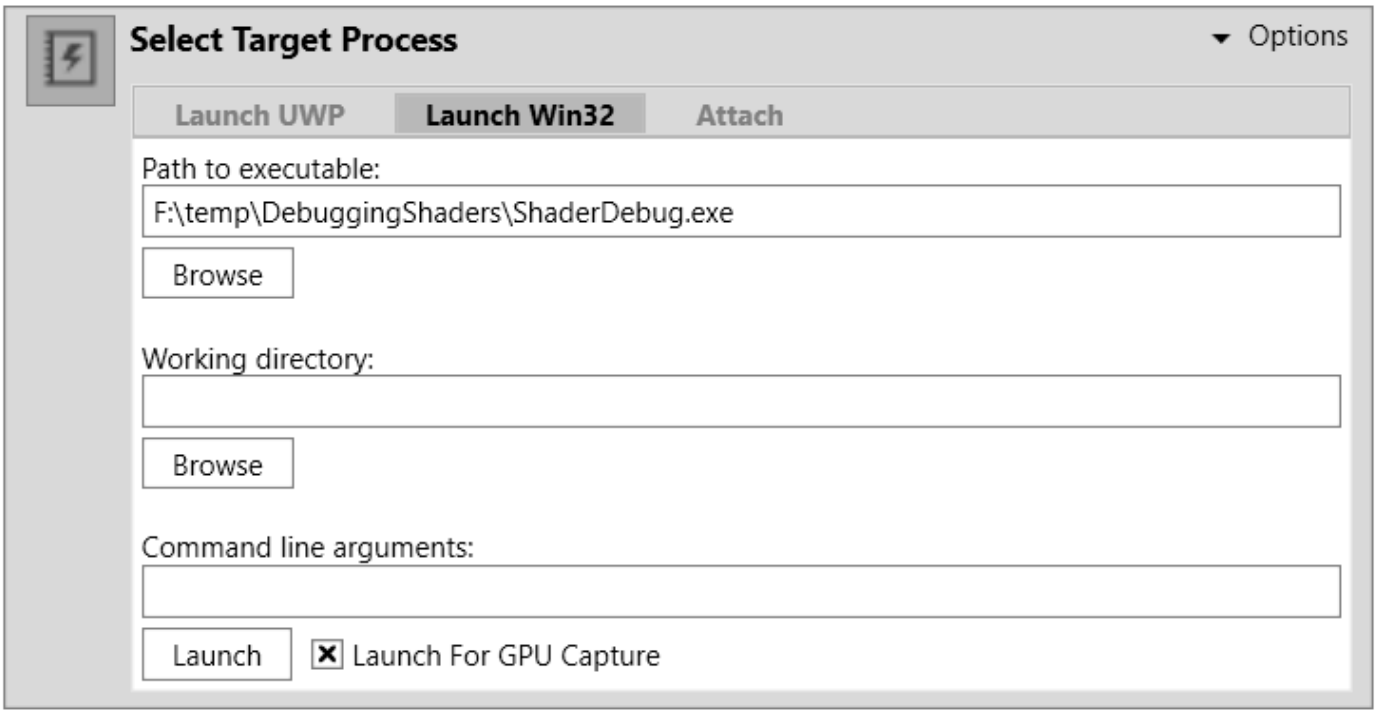
Launch **PIX.**

Click on **Home**, then **Connect**

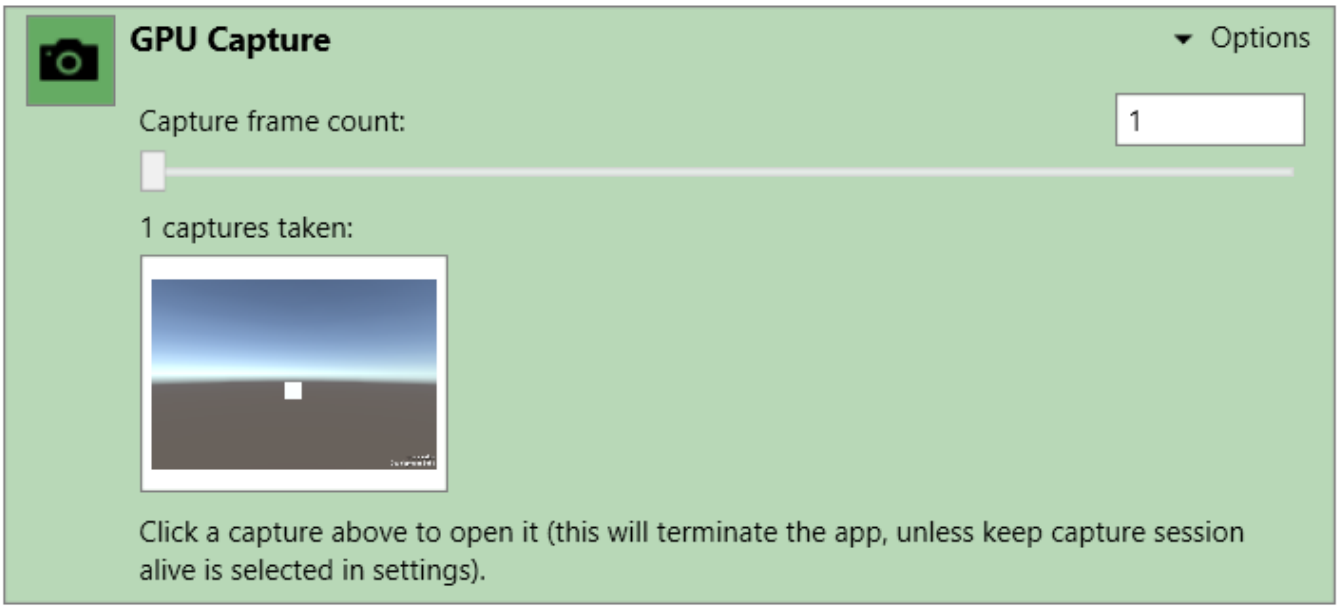
Select Computer **localhost** to use your PC for capturing, and click **connect**.

In the **Select Target Process** box, select the **Launch Win32** tab and use the **Browse** button to select your application's executable file. Note that here, "Win32" means a non-UWP application; your application file must be a 64-bit binary file.

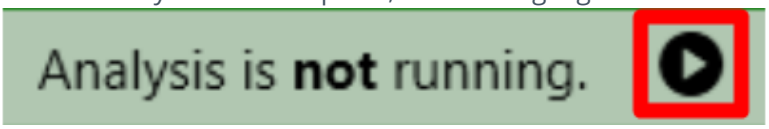
Enable **Launch for GPU Capture**, then use the **Launch** button to start the application.



Use your application as normal until you are ready to capture a frame. To capture a frame, press **Print Screen** on your keyboard, or click the **camera** icon on the GPU Capture Panel. A thumbnail of the capture appears in the panel. To open the capture, click the thumbnail.



To start analysis on the capture, click the highlighted text or the small **Play** icon on the menu bar.



Select the **Pipeline** tab and use the__ Events__ window to navigate to a draw call you are interested in.

Overview | Pipeline | Tools | Debug

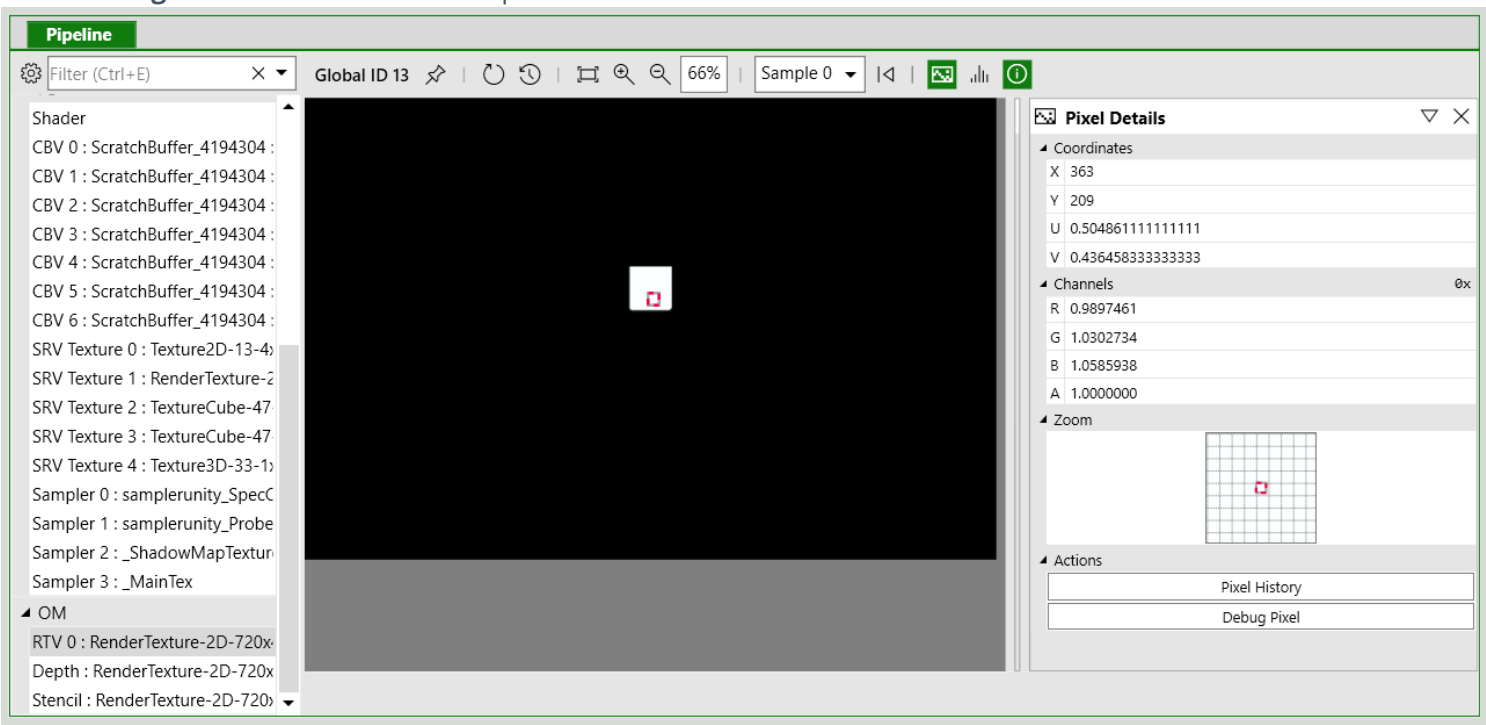
Events

Graphics Queue 0 (Graphics CommandQueue) | .* Aa !G Filter (Ctrl+E)

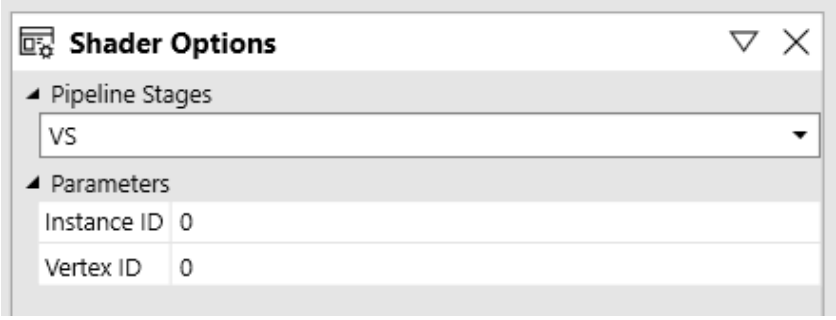
Queue ID	Name	Global ID
0	**Signal**(obj#5,237) {this->ID3D12CommandQueue obj#6,r	1
2	**ResourceBarrier**(6,...) {this->ID3D12GraphicsCommandl	2
7	Camera.Render	3-18
8	UpdateDepthTexture	3-4
23	Drawing	5-14
24	Render.OpaqueGeometry	5-13
26	RenderForwardOpaque.Render	5-13
27	Shadows.RenderShadowMap	5-7
55	RenderForwardOpaque.CollectShadows	8-10
71	Clear	11
74	RenderForward.RenderLoopJob	12-13
80	**ResourceBarrier**(1,...) {this->ID3D12Graphi	12
85	**DrawIndexedInstanced**(36,1,0,0,0) {this->ID	13
86	Camera.RenderSkybox	14
99	Camera.ImageEffects	15-18
116	GUITexture.Draw	19
126	GUITexture.Draw	20

In the lower half of the **Pipeline** tab, select a render target from the **OM** (Output Merger) list to view the output of draw call. Select a **pixel** on the object you want to debug. Note that you can right-click a **pixel** to view the draw call history, as a way of finding draw calls you are interested in.

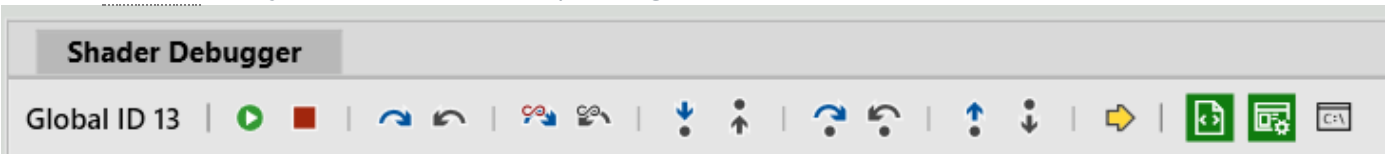
Select **Debug Pixel** on the **Pixel Details** panel.



On the debug panel, use the Shader Options to select which shader stage to debug.



Use the **toolbar** or keyboard shortcuts to step through the code.



For more information on debugging shaders with PIX, see Microsoft's video series PIX on Windows, particularly Part 5 - Debug Tab.

For more information on GPU capture in PIX, see Microsoft's documentation on GPU Captures.

2018–09–17 Page published with editorial review

Implementing Fixed Function TexGen in Shaders

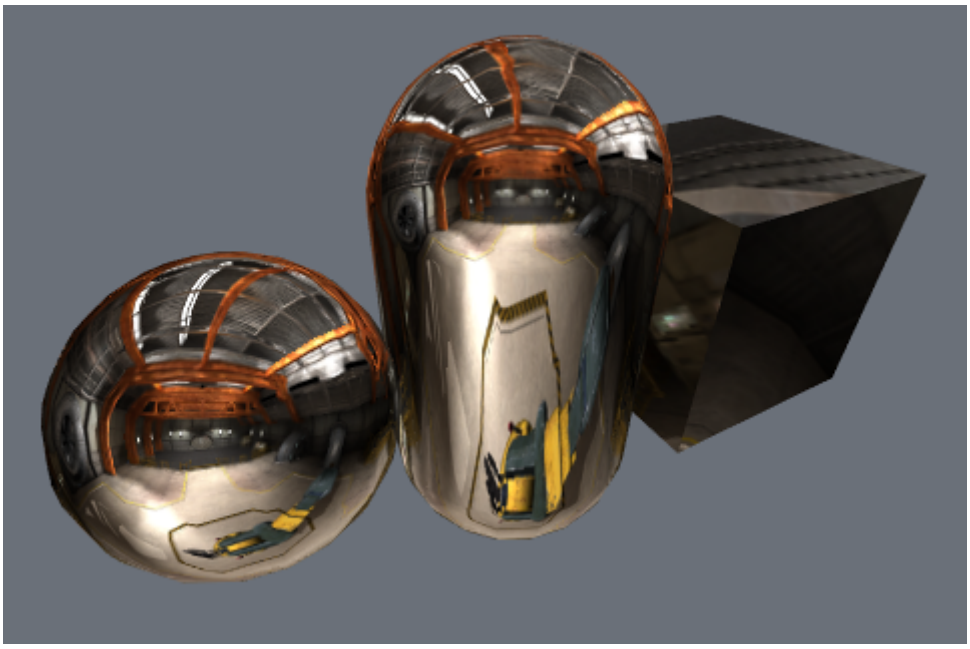
Leave feedback

Before Unity 5, texture properties could have options inside the curly brace block, e.g. `TexGen CubeReflect`. These were controlling fixed function texture coordinate generation. This functionality was removed in Unity 5.0; if you need texgen you should write a vertex shader instead.

This page shows how to implement each of fixed function TexGen modes from Unity 4.

Cubemap reflection (TexGen CubeReflect)

`TexGen CubeReflect` is typically used for simple cubemap reflections. It reflects view direction along the normal in view space, and uses that as the UV coordinate.

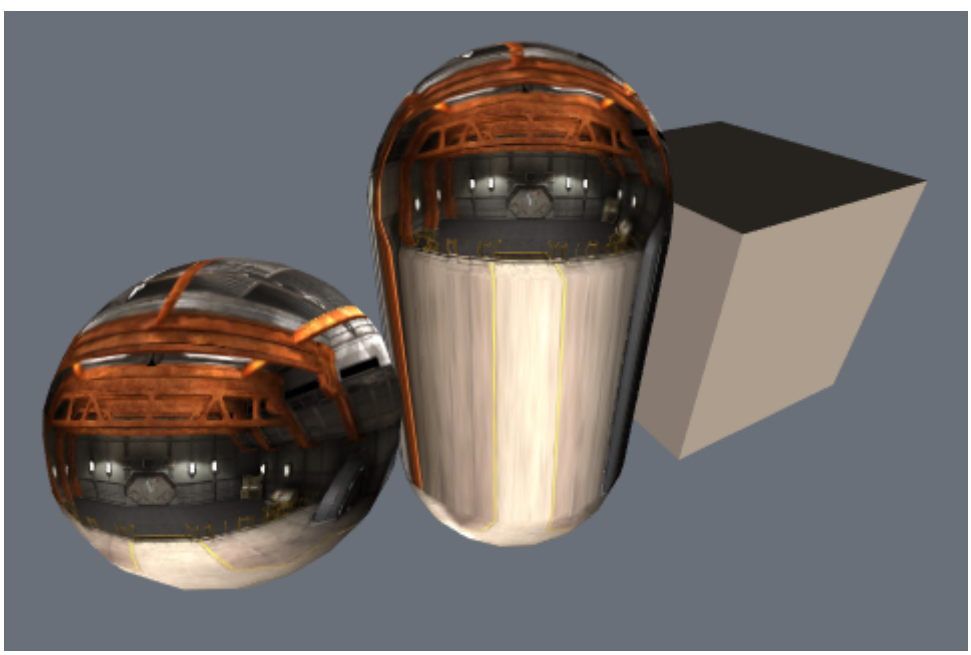


```
Shader "TexGen/CubeReflect" {
Properties {
    _Cube ("Cubemap", Cube) = "" { /* used to be TexGen CubeReflect */ }
}
SubShader {
    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        struct v2f {
            float4 pos : SV_POSITION;
            float3 uv : TEXCOORD0;
        };
```

```
        v2f vert (float4 v : POSITION, float3 n : NORMAL)
        {
            v2f o;
            o.pos = UnityObjectToClipPos(v);

            // TexGen CubeReflect:
            // reflect view direction along the normal,
            // in view space
            float3 viewDir = normalize(ObjSpaceViewDir(v));
            o.uv = reflect(-viewDir, n);
            o.uv = mul(UNITY_MATRIX_MV, float4(o.uv,0));
            return o;
        }

        samplerCUBE _Cube;
        half4 frag (v2f i) : SV_Target
        {
            return texCUBE(_Cube, i.uv);
        }
        ENDCG
    }
}
}
```

Cubemap normal (TexGen CubeNormal)

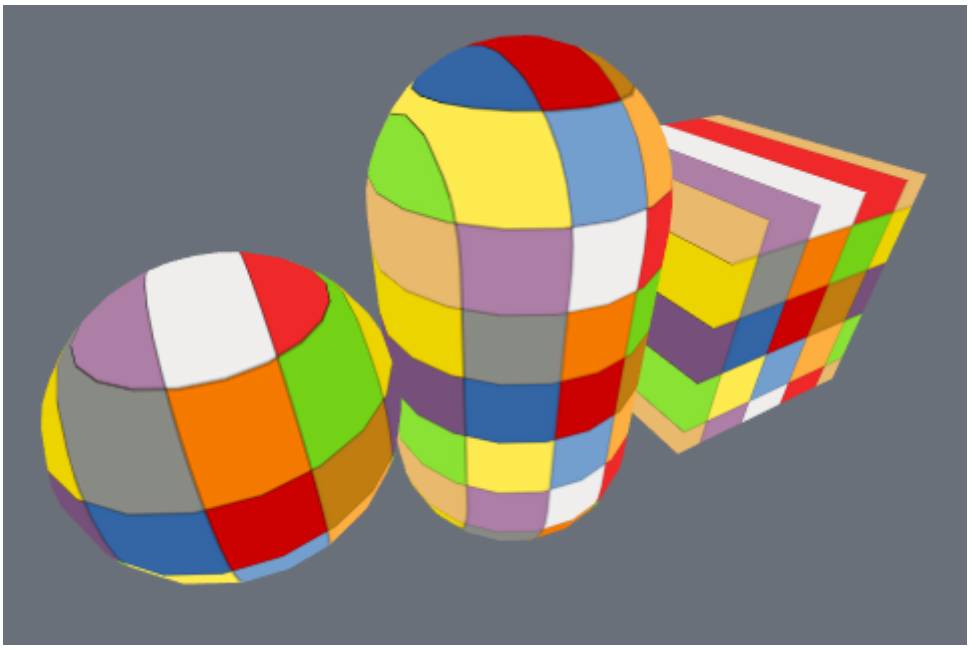
`TexGen CubeNormal` is typically used with cubemaps too. It uses view space normal as the UV coordinate.



```
Shader "TexGen/CubeNormal" {
Properties {
    _Cube ("Cubemap", Cube) = "" { /* used to be TexGen CubeNormal */ }
}
SubShader {
    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        struct v2f {
            float4 pos : SV_POSITION;
            float3 uv : TEXCOORD0;
        };

        v2f vert (float4 v : POSITION, float3 n : NORMAL)
        {
            v2f o;
            o.pos = UnityObjectToClipPos(v);

            // TexGen CubeNormal:
            // use view space normal of the object
            o.uv = mul((float3x3)UNITY_MATRIX_IT_MV, n);
            return o;
        }

        samplerCUBE _Cube;
        half4 frag (v2f i) : SV_Target
        {
            return texCUBE(_Cube, i.uv);
        }
        ENDCG
    }
}
}
```

Object space coordinates (TexGen ObjectLinear)

`TexGen ObjectLinear` used object space vertex position as UV coordinate.

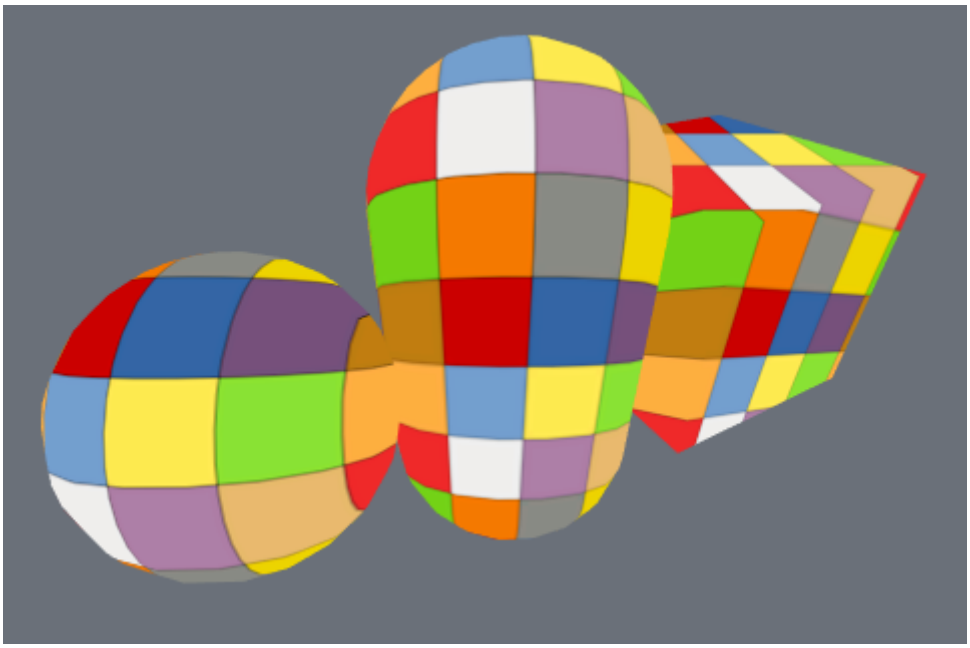


```
Shader "TexGen/ObjectLinear" {
Properties {
    _MainTex ("Texture", 2D) = "" { /* used to be TexGen ObjectLinear */ }
}
SubShader {
    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        struct v2f {
            float4 pos : SV_POSITION;
            float3 uv : TEXCOORD0;
        };

        v2f vert (float4 v : POSITION)
        {
            v2f o;
            o.pos = UnityObjectToClipPos(v);

            // TexGen ObjectLinear:
            // use object space vertex position
            o.uv = v.xyz;
            return o;
        }

        sampler2D _MainTex;
        half4 frag (v2f i) : SV_Target
        {
```

```
            return tex2D(_MainTex, i.uv.xy);
        }
        ENDCG
    }
}
}
```

View space coordinates (TexGen EyeLinear)

`TexGen EyeLinear` used view space vertex position as UV coordinate.

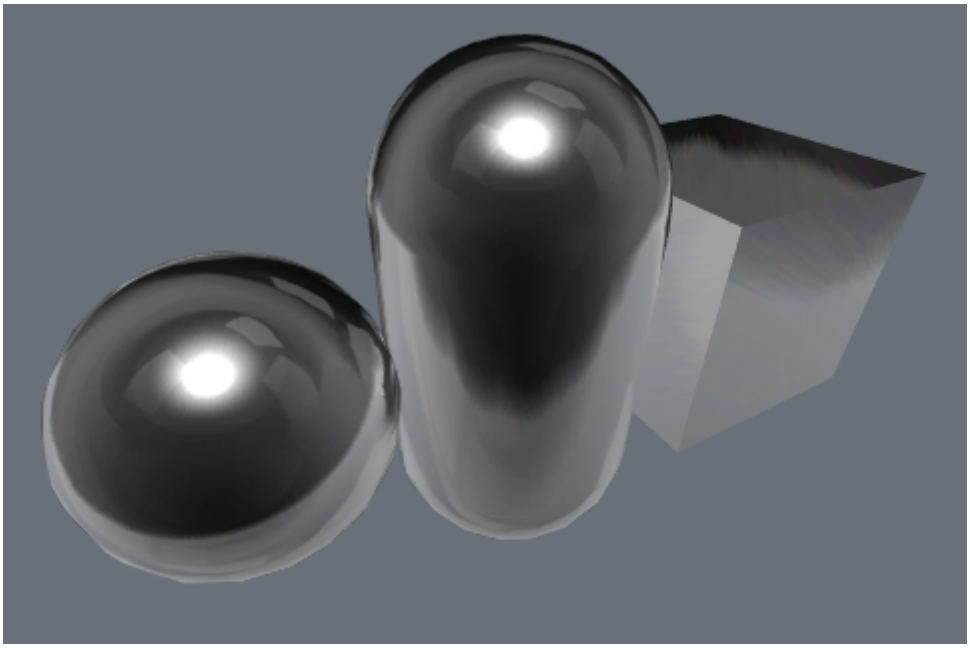


```
Shader "TexGen/EyeLinear" {
Properties {
    _MainTex ("Texture", 2D) = "" { /* used to be TexGen EyeLinear */ }
}
SubShader {
    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        struct v2f {
            float4 pos : SV_POSITION;
            float3 uv : TEXCOORD0;
        };

        v2f vert (float4 v : POSITION)
```

```
        {
            v2f o;
            o.pos = UnityObjectToClipPos(v);

            // TexGen EyeLinear:
            // use view space vertex position
            o.uv = UnityObjectToViewPos(v);
            return o;
        }

        sampler2D _MainTex;
        half4 frag (v2f i) : SV_Target
        {
            return tex2D(_MainTex, i.uv.xy);
        }
        ENDCG
    }
}
}
```

Spherical environment mapping (TexGen SphereMap)

`TexGen SphereMap` computes UV coordinates for spherical environment mapping. See OpenGL TexGen reference for the formula.



```
Shader "TexGen/SphereMap" {
Properties {
    _MainTex ("Texture", 2D) = "" { /* used to be TexGen SphereMap */ }
```

```
}
SubShader {
    Pass {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        struct v2f {
            float4 pos : SV_POSITION;
            float2 uv : TEXCOORD0;
        };

        v2f vert (float4 v : POSITION, float3 n : NORMAL)
        {
            v2f o;
            o.pos = UnityObjectToClipPos(v);

            // TexGen SphereMap
            float3 viewDir = normalize(ObjSpaceViewDir(v));
            float3 r = reflect(-viewDir, n);
            r = mul((float3x3)UNITY_MATRIX_MV, r);
            r.z += 1;
            float m = 2 * length(r);
            o.uv = r.xy / m + 0.5;

            return o;
        }

        sampler2D _MainTex;
        half4 frag (v2f i) : SV_Target
        {
            return tex2D(_MainTex, i.uv);
        }
        ENDCG
    }
}
}
```

Particle Systems reference

Leave feedback

This section contains reference information for the **Particle System component** and each of its numerous modules. There is also a sub-section covering the legacy **particle system** implementation used in earlier versions of Unity.

Particle System

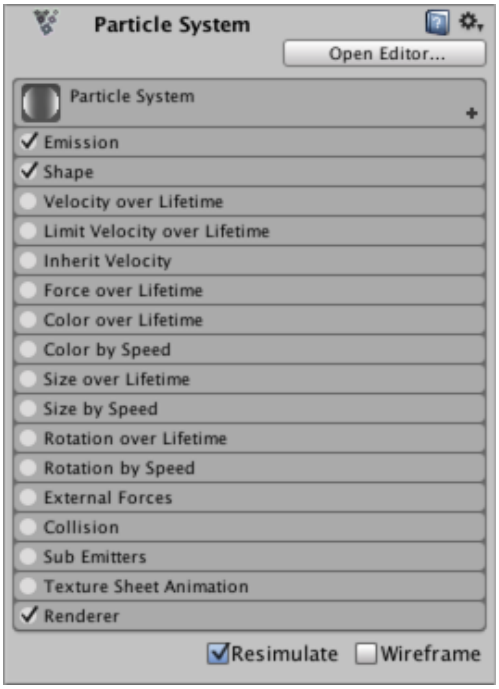
Leave feedback

SWITCH TO SCRIPTING

A **Particle System component** simulates fluid entities such as liquids, clouds and flames by generating and animating large numbers of small 2D images in the **scene**. For a full introduction to particle systems and their uses, see further documentation on Particle Systems.

Properties

The Particle System component has many properties, and for convenience, the **Inspector** organises them into collapsible sections called “modules”. These modules are documented in separate pages. See documentation on Particle System Modules to learn about each one.



To expand and collapse modules, click the bar that shows their name. Use the checkbox on the left to enable or disable the functionality of the properties in that module. For example, if you don’t want to vary the sizes of particles over their lifetime, uncheck the **Size over Lifetime** module.

Aside from the module bars, the Inspector contains a few other controls. The **Open Editor** button shows the options in a separate editor window that also allows you to edit multiple systems at once. The **Resimulate** checkbox determines whether or not property changes should be applied immediately to particles already generated by the system (the alternative is that existing particles are left as they are and only the new particles have the changed properties). The **Selection** button shows the outlines and wireframes of the **Mesh** objects used to show the particles in the Scene, based on the selection mode in the **Gizmos** dropdown. Bounds display the **bounding volume** around the selected Particle Systems. These are used to determine whether a particle System is currently on screen.

Particle System modules

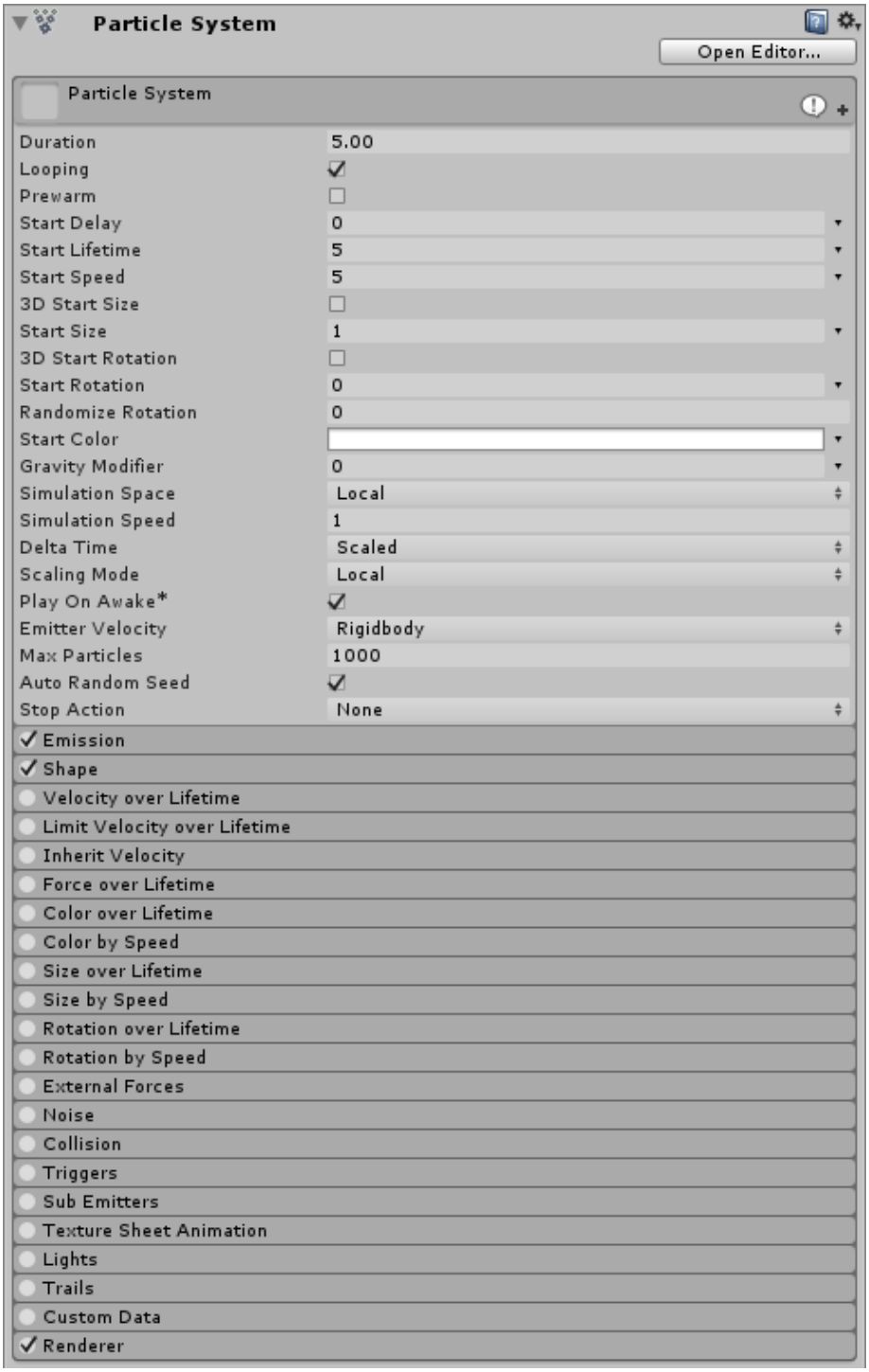
Leave feedback

The Particle System **component** has a powerful set of properties that are organized into modules for ease of use. This section of the manual covers each of the modules in detail.

Particle System Main module

Leave feedback

The **Particle System** module contains global properties that affect the whole system. Most of these properties control the initial state of newly created particles. To expand and collapse the main module, click the **Particle System** bar in the **Inspector** window.



The name of the module appears in the inspector as the name of the **GameObject** that the Particle System component is attached to.

Properties

Property	Function
Duration	The length of time the system runs.

Property	Function
Looping	If enabled, the system starts again at the end of its duration time and continues to repeat the cycle.
Prewarm	If enabled, the system is initialized as though it had already completed a full cycle (only works if **Looping** is also enabled).
Start Delay	Delay in seconds before the system starts emitting once enabled.
Start Lifetime	The initial lifetime for particles.
Start Speed	The initial speed of each particle in the appropriate direction.
3D Start Size	Enable this if you want to control the size of each axis separately.
Start Size	The initial size of each particle.
3D Start Rotation	Enable this if you want to control the rotation of each axis separately.
Start Rotation	The initial rotation angle of each particle.
Randomize Rotation Direction	Causes some particles to spin in the opposite direction.
Start Color	The initial color of each particle.
Gravity Modifier	Scales the gravity value set in the physics manager. A value of zero switches gravity off.
Simulation Space	Controls whether particles are animated in the parent object's local space (therefore moving with the parent object), in the world space, or relative to a custom object (moving with a custom object of your choosing).
Simulation Speed	Adjust the speed at which the entire system updates.
Delta Time	Choose between **Scaled** and **Unscaled**, where **Scaled** uses the **Time Scale** value in the **Time Manager**, and **Unscaled** ignores it. This is useful for Particle Systems that appear on a Pause Menu, for example.
Scaling Mode	Choose how to use the scale from the transform. Set to **Hierarchy**, **Local** or **Shape**. Local applies only the Particle System transform scale, ignoring any parents. Shape mode applies the scale to the start positions of the particles, but does not affect their size.
Play on Awake	If enabled, the Particle System starts automatically when the object is created.
Emitter Velocity	Choose how the Particle System calculates the velocity used by the Inherit Velocity and Emission modules. The system can calculate the velocity using a **Rigidbody component**, if one exists, or by tracking the movement of the Transform **component**.
Max Particles	The maximum number of particles in the system at once. If the limit is reached, some particles are removed.
Auto Random Seed	If enabled, the Particle System looks different each time it is played. When set to false, the system is exactly the same every time it is played.
Random Seed	When disabling the automatic random seed, this value is used to create a unique repeatable effect.

Property	Function
Stop Action	When all the particles belonging to the system have finished, it is possible to make the system perform an action. A system is determined to have stopped when all its particles have died, and its age has exceeded its Duration. For looping systems, this only happens if the system is stopped via script.
Disable	The GameObject is disabled.
Destroy	The GameObject is destroyed.
Callback	The OnParticleSystemStopped callback is sent to any **scripts** attached to the GameObject.

Property details

The system emits particles for a specific duration, and can be set to emit continuously using the **Looped** property. This allows you to set particles to be emitted intermittently or continuously; for example, an object may emit smoke in short puffs or in a steady stream.

The **Start** properties (**lifetime**, **speed**, **size**, **rotation** and **color**) specify the state of a particle on emission. You can specify a particle's width, height and depth independently, using the **3D Start Size** property (see Non-uniform particle scaling, below).

All Particle Systems use the same gravity vector specified in the **Physics** settings. The Gravity Multiplier value can be used to scale the gravity, or switch it off if set to zero.

The **Simulation Space** property determines whether the particles move with the Particle System parent object, a custom object, or independently in the game world. For example, systems like clouds, hoses and flamethrowers need to be set independently of their parent GameObject, as they tend to leave trails that persist in the world space even if the object producing them moves around. On the other hand, if particles are used to create a spark between two electrodes, the particles should move along with the parent object. For more advanced control over how particles follow their Transform, see documentation on the Inherit Velocity module.

Non-uniform particle scaling

The 3D Start Size property allows you to specify a particle's width, height and depth independently. In the Particle System **Main** module, check the **3D Start Size** checkbox, and enter the values for the initial x (width), y (height) and z (depth) of the particle. Note that z (depth) only applies to 3D **Mesh** particles. You can also set randomised values for these properties, in a range between two constants or curves.

You can set the particle's initial size in the Particle System **Main** module, and its size over the particle's lifetime using the **Separate Axes** option in the **Size over Lifetime** module. You can also set the particle's size in relation to its speed using the **Separate Axes** option in the **Size by Speed** module.

2017–05–31 Page amended with editorial review

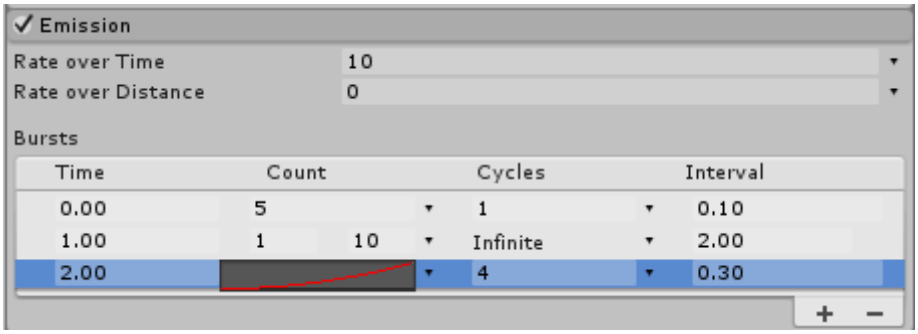
Simulation Speed, Delta Time and Emitter Velocity added in Unity 2017.1

Stop Action particle system property added in Unity 2017.2

Emission module

Leave feedback

The properties in this module affect the rate and timing of Particle System emissions.



Properties

Property	Function
Rate over Time	The number of particles emitted per unit of time.
Rate over Distance	The number of particles emitted per unit of distance moved.
Bursts	A burst is an event which spawns particles. These settings allow particles to be emitted at specified times.
Time	Set the time (in seconds, after the Particle System begins playing) at which to emit the burst.
Count	Set a value for the number of particles that may be emitted.
Cycles	Set a value for how many times to play the burst.
Interval	Set a value for the time (in seconds) between when each cycle of the burst is triggered.

Details

The rate of emission can be constant or can vary over the lifetime of the system according to a curve. If **Rate over Distance** mode is active, a certain number of particles are released per unit of distance moved by the parent object. This is very useful for simulating particles that are actually created by the motion of the object (for example, dust from a car's wheels on a dirt track).

If **Rate over Time** is active, then the desired number of particles are emitted each second regardless of how the parent object moves. Additionally, you can add bursts of extra particles that appear at specific times (for example, a steam train chimney that produces puffs of smoke).

Particle System Shape Module

Leave feedback

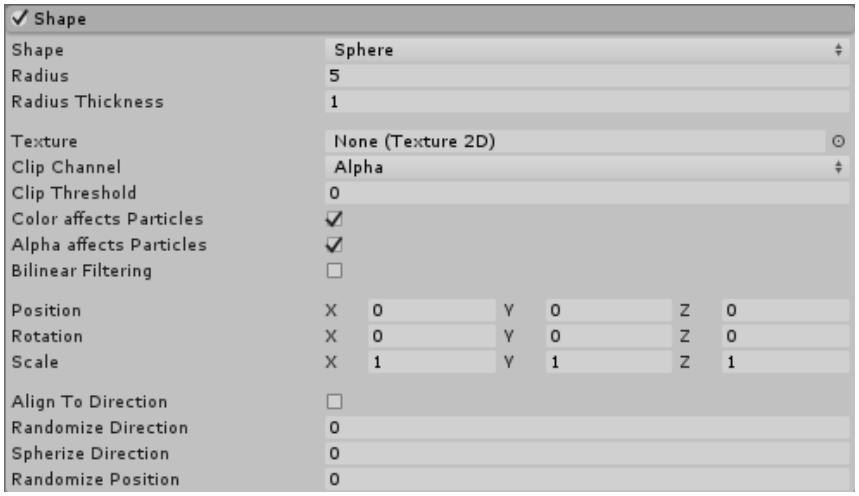
This module defines the the volume or surface from which particles can be emitted, and the direction of the start velocity. The **Shape** property defines the shape of the emission volume, and the rest of the module properties vary depending on the Shape you choose.

All shapes (except Mesh) have properties that define their dimensions, such as the **Radius** property. To edit these, drag the handles on the wireframe emitter shape in the **Scene view**. The choice of shape affects the region from which particles can be launched, but also the initial direction of the particles. For example, a **Sphere** emits particles outward in all directions, a **Cone** emits a diverging stream of particles, and a **Mesh** emits particles in directions that are normal to the surface.

The section below details the properties for each **Shape**.

Shapes in the Shape module

Sphere, Hemisphere



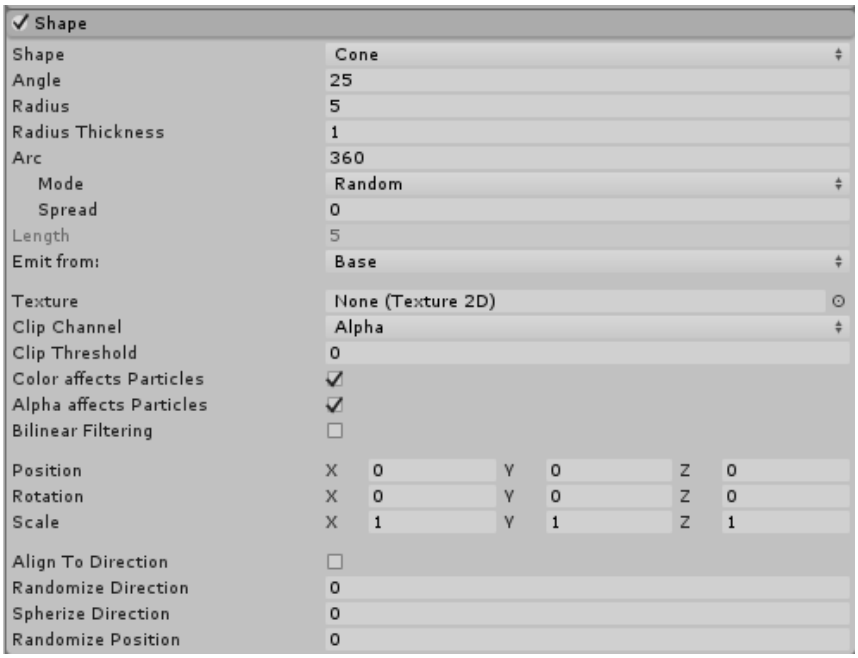
The shape module when set to Sphere mode

Note: Sphere and Hemisphere have the same properties.

Property	Function
Shape	The shape of the emission volume.
Sphere	Uniform particle emission in all directions.
Hemisphere	Uniform particle emission in all directions on one side of a plane.
Radius	The radius of the circular aspect of the shape.
Radius Thickness	The proportion of the volume that emits particles. A value of 0 emits particles from the outer surface of the shape. A value of 1 emits particles from the entire volume. Values in between will use a proportion of the volume.
Texture	A texture to use for tinting and discarding particles.
Clip Channel	A channel from the texture to use for discarding particles.
Clip Threshold	When mapping particles to positions on the texture, discard any whose **pixel** color falls below this threshold.
Color affects Particles	Multiply particle colors by the texture color.
Alpha affects Particles	Multiply particle alphas by the texture alpha.

Property	Function
Bilinear Filtering	When reading the texture, combine 4 neighboring samples for smoother changes in particle color, regardless of the texture dimensions.
Position	Apply an offset to the emitter shape used for spawning particles.
Rotation	Rotate the emitter shape used for spawning particles.
Scale	Change the size of the emitter shape used for spawning particles.
Align to Direction	Orient particles based on their initial direction of travel. This can be useful if you want to simulate, for example, chunks of car paint flying off a car's bodywork during a **collision**. If the orientation is not satisfactory, you can also override it by applying a **Start Rotation** value in the **Main** module.
Randomize Direction	Blend particle directions towards a random direction. When set to 0, this setting has no effect. When set to 1, the particle direction is completely random.
Spherize Direction	Blend particle directions towards a spherical direction, where they travel outwards from the center of their Transform. When set to 0, this setting has no effect. When set to 1, the particle direction points outwards from the center (behaving identically to when the Shape is set to Sphere).
Randomize Position	Move particles by a random amount, up to the specified value. When this is set to 0, this setting has no effect. Any other value will apply some randomness to the spawning positions of the particles.

Cone

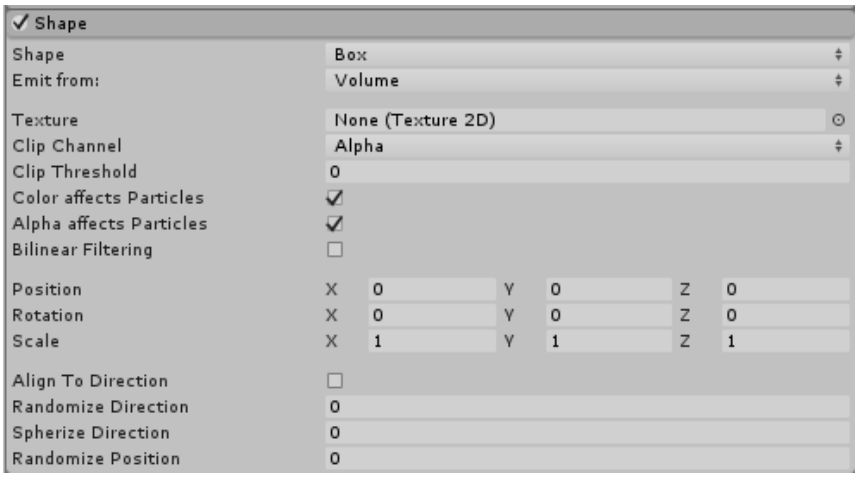


The shape module when set to Cone mode

Property	Function
Shape	The shape of the emission volume.
Cone	Emit particles from the base or body of a cone. The particles diverge in proportion to their distance from the cone's center line.
Angle	The angle of the cone at its point. An angle of 0 produces a cylinder while an angle of 90 gives a flat disc.
Radius	The radius of the circular aspect of the shape.
Radius Thickness	The proportion of the volume that emits particles.A value of 0 emits particles from the outer surface of the shape. A value of 1 emits particles from the entire volume. Values in between will use a proportion of the volume.
Arc	The angular portion of a full circle that forms the emitter's shape.

Property	Function
Mode	Define how Unity generates particles around the arc of the shape. When set to **Random**, Unity generates particles randomly around the arc. If using **Loop**, Unity generates particles sequentially around the arc of the shape, and loops back to the start at the end of each cycle. **Ping-Pong** is the same as **Loop**, except each consecutive loop happens in the opposite direction to the last. Finally, **Burst Spread** mode distributes particle generation evenly around the shape. This can give you an even spread of particles, compared to the default randomized behavior, where particles may clump together unevenly. **Burst Spread** is best used with burst emissions.
Spread	The discrete intervals around the arc where particles may be generated. For example, a value of 0 allows particles to spawn anywhere around the arc, and a value of 0.1 only spawns particles at 10% intervals around the shape.
Speed	The speed the emission position moves around the arc. Using the small black drop-down menu next to the value field, set this to **Constant** for the value to always remain the same, or **Curve** for the value to change over time. This option is only available if **Mode** is set to something other than **Random**
Length	The length of the cone. This only applies when the **Emit from:** property is set to **Volume**.
Emit from:	The part of the cone to emit particles from: **Base** or **Volume**.
Texture	A texture to be used for tinting and discarding particles.
Clip Channel	A channel from the texture to be used for discarding particles.
Clip Threshold	When mapping particles to positions on the texture, discard any whose pixel color falls below this threshold.
Color affects Particles	Multiply particle colors by the texture color.
Alpha affects Particles	Multiply particle alphas by the texture alpha.
Bilinear Filtering	When reading the texture, combine 4 neighboring samples for smoother changes in particle color, regardless of the texture dimensions.
Position	Apply an offset to the emitter shape used for spawning particles.
Rotation	Rotate the emitter shape used for spawning particles.
Scale	Change the size of the emitter shape used for spawning particles.
Align to Direction	Orient particles based on their initial direction of travel. This can be useful if you want to simulate, for example, chunks of car paint flying off a car's bodywork during a collision. If the orientation is not satisfactory, you can also override it by applying a Start Rotation value in the Main module.
Randomize Direction	Blend particle directions towards a random direction. When set to 0, this setting has no effect. When set to 1, the particle direction is completely random.
Spherize Direction	Blend particle directions towards a spherical direction, where they travel outwards from the center of their Transform. When set to 0, this setting has no effect. When set to 1, the particle direction points outwards from the center (behaving identically to when the Shape is set to Sphere).
Randomize Position	Move particles by a random amount, up to the specified value. When this is set to 0, this setting has no effect. Any other value will apply some randomness to the spawning positions of the particles.

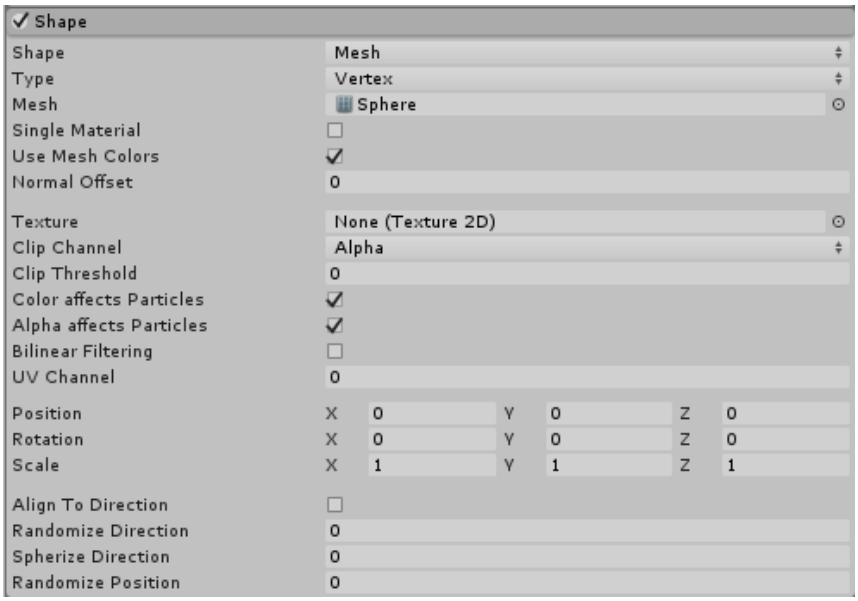
Box



The shape module when set to Box mode

Property	Function
Shape	The shape of the emission volume.
Box	Emit particles from the edge, surface, or body of a box shape. The particles move in the emitter object's forward (Z) direction.
Emit from:	Select the part of the box to emit from: **Edge**, **Shell**, or **Volume**.
Texture	A texture to be used for tinting and discarding particles.
Clip Channel	A channel from the texture to be used for discarding particles.
Clip Threshold	When mapping particles to positions on the texture, discard any whose pixel color falls below this threshold.
Color affects Particles	Multiply particle colors by the texture color.
Alpha affects Particles	Multiply particle alphas by the texture alpha.
Bilinear Filtering	When reading the texture, combine 4 neighboring samples for smoother changes in particle color, regardless of the texture dimensions.
Position	Apply an offset to the emitter shape used for spawning particles.
Rotation	Rotate the emitter shape used for spawning particles.
Scale	Change the size of the emitter shape used for spawning particles.
Align to Direction	Orient particles based on their initial direction of travel. This can be useful if you want to simulate, for example, chunks of car paint flying off a car's bodywork during a collision. If the orientation is not satisfactory, you can also override it by applying a Start Rotation value in the Main module.
Randomize Direction	Blend particle directions towards a random direction. When set to 0, this setting has no effect. When set to 1, the particle direction is completely random.
Spherize Direction	Blend particle directions towards a spherical direction, where they travel outwards from the center of their Transform. When set to 0, this setting has no effect. When set to 1, the particle direction points outwards from the center (behaving identically to when the Shape is set to Sphere).
Randomize Position	Move particles by a random amount, up to the specified value. When this is set to 0, this setting has no effect. Any other value will apply some randomness to the spawning positions of the particles.

Mesh, MeshRenderer, SkinnedMeshRenderer



The shape module when set to Mesh mode

Mesh, MeshRenderer and SkinnedMeshRenderer have the same properties.

Property	Function
Shape	The shape of the emission volume.
Mesh	Emits particles from any arbitrary Mesh shape supplied via the **Inspector**.
MeshRenderer	Emits particles from a reference to a **GameObject**'s **Mesh Renderer**.
SkinnedMeshRenderer	Emits particles from a reference to a GameObject's Skinned Mesh Renderer.
Emission drop-down	Where particles are emitted from. Select **Vertex** for the particles to emit from the vertices, **Edge** for the particles to emit from the edges, or **Triangle** for the particles to emit from the triangles. This is set to **Vertex** by default.
Mesh	The Mesh that provides the emitter's shape.
Single Material	Specify whether to emit particles from a particular sub-Mesh (identified by the material index number). If enabled, a numeric field appears, which allows you to specify the material index number.
Use Mesh Colors	Modulate particle color with Mesh vertex colors, or, if they don't exist, use the **shader** color property "*Color*" *or* "TintColor" from the material.
Normal Offset	Distance away from the surface of the Mesh to emit particles (in the direction of the surface normal)
Texture	A texture to be used for tinting and discarding particles.
Clip Channel	A channel from the texture to be used for discarding particles.
Clip Threshold	When mapping particles to positions on the texture, discard any whose pixel color falls below this threshold.
Color Affects Particles	Multiply particle colors by the texture color.
Alpha Affects Particles	Multiply particle alphas by the texture alpha.
Bilinear Filtering	When reading the texture, combine 4 neighboring samples for smoother changes in particle color, regardless of the texture dimensions.
UV Channel	Choose which UV channel on the source mesh to use for sampling the texture.
Position	Apply an offset to the emitter shape used for spawning particles.
Rotation	Rotate the emitter shape used for spawning particles.
Scale	Change the size of the emitter shape used for spawning particles.
Align to Direction	Orient particles based on their initial direction of travel. This can be useful if you want to simulate, for example, chunks of car paint flying off a car's bodywork during a collision. If the orientation is not satisfactory, you can also override it by applying a Start Rotation value in the Main module.

Property	Function
Randomize Direction	Blend particle directions towards a random direction. When set to 0, this setting has no effect. When set to 1, the particle direction is completely random.
Spherize Direction	Blend particle directions towards a spherical direction, where they travel outwards from the center of their Transform. When set to 0, this setting has no effect. When set to 1, the particle direction points outwards from the center (behaving identically to when the Shape is set to Sphere).
Randomize Position	Move particles by a random amount, up to the specified value. When this is set to 0, this setting has no effect. Any other value will apply some randomness to the spawning positions of the particles.

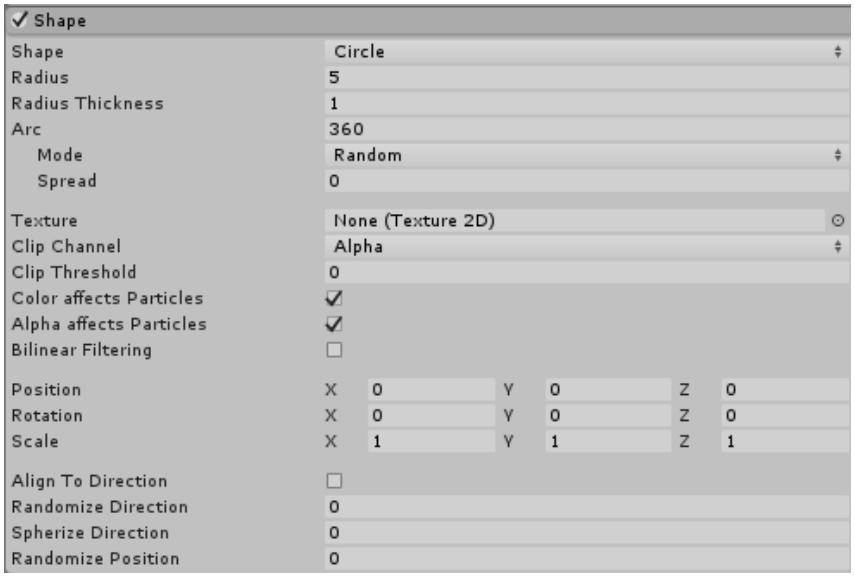
Mesh details

You can choose to only emit particles from a particular material (sub-Mesh) by checking the **Single Material** property and you can offset the emission position along the Mesh's normals by checking the **Normal Offset** property.

To ignore the color of the Mesh, check the **Use Mesh Colors** property. To read the texture colors from a mesh, assign the Texture you wish to read to the **Texture** property.

Meshes must be read/write enabled to work on the **particle system**. If you assign them in the Editor, Unity handles this for you. But if you want to assign different meshes at run time, you need to check the **Read/Write Enabled** setting in the Import Settings.

Circle

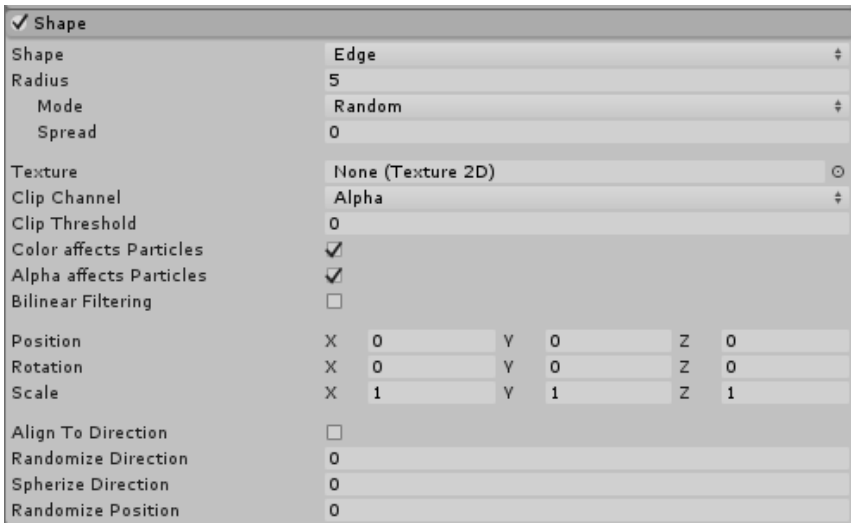


The shape module when set to Circle mode

Property	Function
Shape	The shape of the emission volume.
Circle	Uniform particle emission from the center or edge of a circle. The particles move only in the plane of the circle.
Radius	The radius of the circular aspect of the shape.
Radius Thickness	The proportion of the volume that emits particles. A value of 0 emits particles from the outer edge of the circle. A value of 1 emits particles from the entire area. Values in between will use a proportion of the area.
Arc	The angular portion of a full circle that forms the emitter's shape.

Property	Function
Mode	Define how Unity generates particles around the arc of the shape. When set to **Random**, Unity generates particles randomly around the arc. If using **Loop**, Unity generates particles sequentially around the arc of the shape, and loops back to the start at the end of each cycle. **Ping-Pong** is the same as **Loop**, except each consecutive loop happens in the opposite direction to the last. Finally, **Burst Spread** mode distributes particle generation evenly around the shape. This can be used to give you an even spread of particles, compared to the default randomized behavior, where particles may clump together unevenly. Burst Spread is best used with burst emissions.
Spread	Control the discrete intervals around the arc where particles may be generated. For example, a value of 0 will allow particles to spawn anywhere around the arc, and a value of 0.1 will only spawn particles at 10% intervals around the shape.
Speed	Set a value for the speed the emission position moves around the arc. Using the small black drop-down next to the value field, set this to **Constant** for the value to always remain the same, or **Curve** for the value to change over time.
Texture	Choose a texture to be used for tinting and discarding particles.
Clip Channel	Select a channel from the texture to be used for discarding particles.
Clip Threshold	When mapping particles to positions on the texture, discard any whose pixel color falls below this threshold.
Color Affects Particles	Multiply particle colors by the texture color.
Alpha Affects Particles	Multiply particle alphas by the texture alpha.
Bilinear Filtering	When reading the texture, combine 4 neighboring samples, for smoother changes in particle color, regardless of the texture dimensions.
Position	Apply an offset to the emitter shape used for spawning particles.
Rotation	Rotate the emitter shape used for spawning particles.
Scale	Change the size of the emitter shape used for spawning particles.
Align to Direction	Use this checkbox to orient particles based on their initial direction of travel. This can be useful if you want to simulate, for example, chunks of car paint flying off a car's bodywork during a collision. If the orientation is not satisfactory, you can also override it by applying a Start Rotation value in the Main module.
Randomize Direction	Blend particle directions towards a random direction. When set to 0, this setting has no effect. When set to 1, the particle direction is completely random.
Spherize Direction	Blend particle directions towards a spherical direction, where they travel outwards from the center of their Transform. When set to 0, this setting has no effect. When set to 1, the particle direction points outwards from the center (behaving identically to when the Shape is set to Sphere).
Randomize Position	Move particles by a random amount, up to the specified value. When this is set to 0, this setting has no effect. Any other value will apply some randomness to the spawning positions of the particles.

Edge

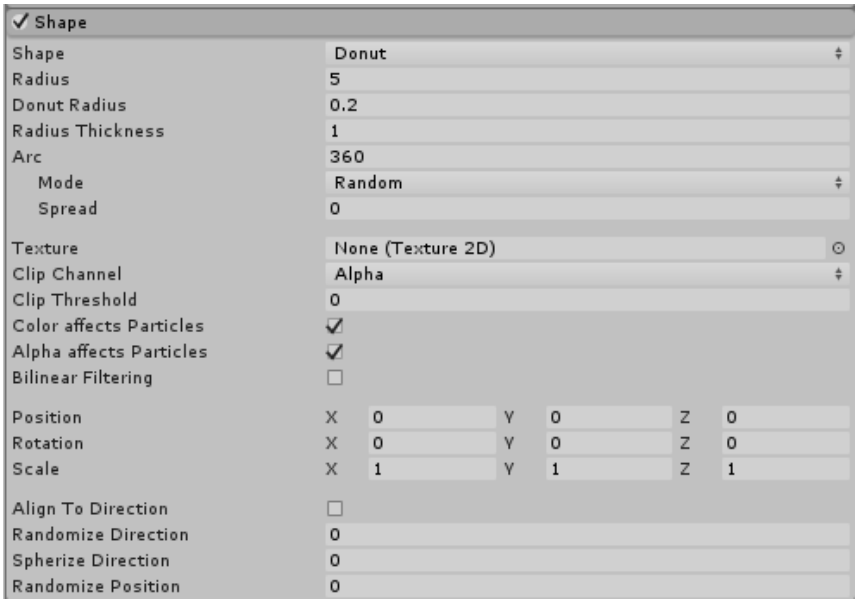


The shape module when set to Edge mode

Property	Function
Shape	The shape of the emission volume.
Edge	Emit particles from a line segment. The particles move in the emitter object's upward (Y) direction.
Radius	The radius property is used to define the length of the edge.
Mode	Define how Unity generates particles along the radius of the shape. When set to **Random**, Unity generates particles randomly along the radius. If using **Loop**, Unity generates particles sequentially along the radius of the shape, and loops back to the start at the end of each cycle. **Ping-Pong** is the same as **Loop**, except each consecutive loop happens in the opposite direction to the last. Finally, **Burst Spread** mode distributes particle generation evenly along the radius. This can be used to give you an even spread of particles, compared to the default randomized behavior, where particles may clump together unevenly. Burst Spread is best used with burst emissions.
Spread	T the discrete intervals along the radius where particles may be generated. For example, a value of 0 will allow particles to spawn anywhere along the radius, and a value of 0.1 will only spawn particles at 10% intervals along the radius.
Speed	The speed the emission position moves along the radius. Using the small black drop-down next to the value field, set this to **Constant** for the value to always remain the same, or **Curve** for the value to change over time.
Texture	A texture to be used for tinting and discarding particles.
Clip Channel	A channel from the texture to be used for discarding particles.
Clip Threshold	When mapping particles to positions on the texture, discard any whose pixel color falls below this threshold.
Color Affects Particles	Multiply particle colors by the texture color.
Alpha Affects Particles	Multiply particle alphas by the texture alpha.
Bilinear Filtering	When reading the texture, combine 4 neighboring samples for smoother changes in particle color, regardless of the texture dimensions.
Position	Apply an offset to the emitter shape used for spawning particles.
Rotation	Rotate the emitter shape used for spawning particles.
Scale	Change the size of the emitter shape used for spawning particles.
Align to Direction	Orient particles based on their initial direction of travel. This can be useful if you want to simulate, for example, chunks of car paint flying off a car's bodywork during a collision. If the orientation is not satisfactory, you can also override it by applying a **Start Rotation** value in the Main module.

Property	Function
Randomize Direction	Blend particle directions towards a random direction. When set to 0, this setting has no effect. When set to 1, the particle direction is completely random.
Spherize Direction	Blend particle directions towards a spherical direction, where they travel outwards from the center of their Transform. When set to 0, this setting has no effect. When set to 1, the particle direction points outwards from the center (behaving identically to when the Shape is set to Sphere).
Randomize Position	Move particles by a random amount, up to the specified value. When this is set to 0, this setting has no effect. Any other value will apply some randomness to the spawning positions of the particles.

Donut

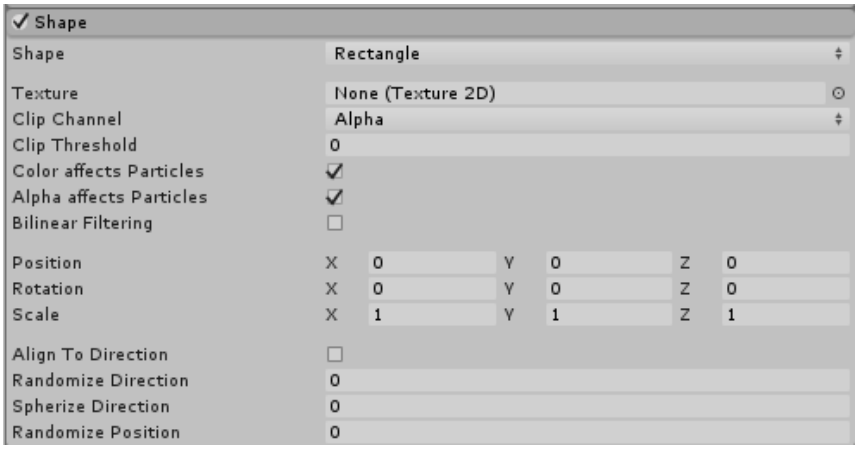


The shape module when set to Donut mode

Property	Function
Shape	The shape of the emission volume.
Donut	Emit particles from a torus. The particles move outwards from the ring of the Torus.
Radius	The radius of the main donut ring.
Donus Radius	The thickness of the outer donut ring.
Radius Thickness	The proportion of the volume that emits particles. A value of 0 emits particles from the outer edge of the circle. A value of 1 emits particles from the entire area. Values in between will use a proportion of the area.
Arc	The angular portion of a full circle that forms the emitter's shape.
Mode	Define how Unity generates particles around the arc of the shape. When set to **Random**, Unity generates particles randomly around the arc. If using **Loop**, Unity generates particles sequentially around the arc of the shape, and loops back to the start at the end of each cycle. **Ping-Pong** is the same as **Loop**, except each consecutive loop happens in the opposite direction to the last. Finally, **Burst Spread** mode distributes particle generation evenly around the shape. This can be used to give you an even spread of particles, compared to the default randomized behavior, where particles may clump together unevenly. Burst Spread is best used with burst emissions.
Spread	The discrete intervals around the arc where particles may be generated. For example, a value of 0 will allow particles to spawn anywhere around the arc, and a value of 0.1 will only spawn particles at 10% intervals around the shape.
Speed	The speed the emission position moves around the arc. Using the small black drop-down next to the value field, set this to **Constant** for the value to always remain the same, or **Curve** for the value to change over time.
Texture	A texture to be used for tinting and discarding particles.

Property	Function
Clip Channel	A channel from the texture to be used for discarding particles.
Clip Threshold	When mapping particles to positions on the texture, discard any whose pixel color falls below this threshold.
Color Affects Particles	Multiply particle colors by the texture color.
Alpha Affects Particles	Multiply particle alphas by the texture alpha.
Bilinear Filtering	When reading the texture, combine 4 neighboring samples for smoother changes in particle color, regardless of the texture dimensions.
Position	Apply an offset to the emitter shape used for spawning particles.
Rotation	Rotate the emitter shape used for spawning particles.
Scale	Change the size of the emitter shape used for spawning particles.
Align To Direction	Orient particles based on their initial direction of travel. This can be useful if you want to simulate, for example, chunks of car paint flying off a car's bodywork during a collision. If the orientation is not satisfactory, you can also override it by applying a **Start Rotation** value in the Main module.
Randomize Direction	Blend particle directions towards a random direction. When set to 0, this setting has no effect. When set to 1, the particle direction is completely random.
Spherize Direction	Blend particle directions towards a spherical direction, where they travel outwards from the center of their Transform. When set to 0, this setting has no effect. When set to 1, the particle direction points outwards from the center (behaving identically to when the Shape is set to Sphere).
Randomize Position	Move particles by a random amount, up to the specified value. When this is set to 0, this setting has no effect. Any other value will apply some randomness to the spawning positions of the particles.

Rectangle



The shape module when set to Rectangle mode

Property	Function
Shape	The shape of the emission volume.
Rectangle	Emits particles from a rectangle. The particles move up from the rectangle.
Texture	A texture to be used for tinting and discarding particles.
Clip Channel	A channel from the texture to be used for discarding particles.
Clip Threshold	When mapping particles to positions on the texture, discard any whose pixel color falls below this threshold.
Color Affects Particles	Multiply particle colors by the texture color.

Property	Function
Alpha Affects Particles	Multiply particle alphas by the texture alpha.
Bilinear Filtering	When reading the texture, combine 4 neighboring samples for smoother changes in particle color, regardless of the texture dimensions.
Position	Apply an offset to the emitter shape used for spawning particles.
Rotation	Rotate the emitter shape used for spawning particles.
Scale	Change the size of the emitter shape used for spawning particles.
Align To Direction	Orient particles based on their initial direction of travel. This can be useful if you want to simulate, for example, chunks of car paint flying off a car's bodywork during a collision. If the orientation is not satisfactory, you can also override it by applying a **Start Rotation** value in the Main module.
Randomize Direction	Blend particle directions towards a random direction. When set to 0, this setting has no effect. When set to 1, the particle direction is completely random.
Spherize Direction	Blend particle directions towards a spherical direction, where they travel outwards from the center of their Transform. When set to 0, this setting has no effect. When set to 1, the particle direction points outwards from the center (behaving identically to when the Shape is set to Sphere).
Randomize Position	Move particles by a random amount, up to the specified value. When this is set to 0, this setting has no effect. Any other value will apply some randomness to the spawning positions of the particles.

2018–03–28 Page amended with no editorial review

Functionality of Shape Module updated in Unity 2017.1

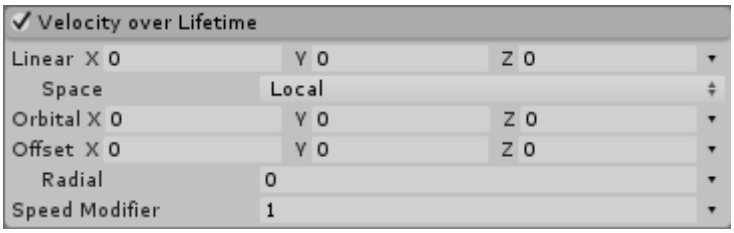
Texture tinting and selective discarding features (Clip Channel, Clip Threshold, Color affects particles, Alpha affects particles, Bilinear filtering) added to Shape Module in 2018.1

Rectangle emission shape added to Shape Module in 2018.1

Velocity over Lifetime module

Leave feedback

The **Velocity over Lifetime** module allows you to control the velocity of particles over their lifetime.



Properties

Property	Function
Linear X, Y, Z	Linear velocity of particles in the X, Y and Z axes.
Space	Specifies whether the **Linear X, Y, Z** axes refer to local or world space.
Orbital X, Y, Z	Orbital velocity of particles around the X, Y and Z axes.
Offset X, Y, Z	The position of the center of orbit, for orbiting particles.
Radial	Radial velocity of particles away from/towards the center position.
Speed Modifier	Applies a multiplier to the speed of particles, along/around their current direction of travel.

Details

To create particles that drift in a particular direction, use the Linear X, Y and Z curves.

To create effects with particles that spin around a center position, use the **Orbital** velocity values. Additionally, you can propel particles towards or away from a center position using the **Radial** velocity values. You can define a custom center of rotation for each particle by using the **Offset** value.

You can also use this module to adjust the speed of the particles in the **Particle System** without affecting their direction, by leaving all the above values at zero and only modifying the **Speed Modifier** value.

2018–03–28 Page amended with limited editorial review

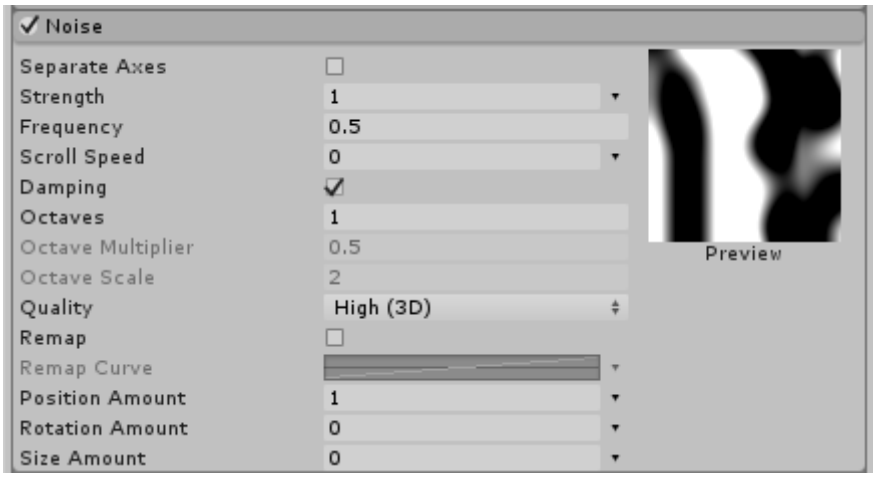
Speed Modifier property added to Velocity over Lifetime module in 2017.3

Orbital XYZ, Offset XYZ and Radial properties added to Velocity over Lifetime module in 2018.1

Noise module

Leave feedback

Add turbulence to particle movement using this module.



Properties

Property	Function
Separate Axes	Control the strength and remapping independently on each axis.
Strength	A curve that defines how strong the noise effect is on a particle over its lifetime. Higher values will make particles move faster and further.
Frequency	Low values create soft, smooth noise, and high values create rapidly changing noise. This controls how often the particles change their direction of travel, and how abrupt those changes of direction are.
Scroll Speed	Move the noise field over time to cause more unpredictable and erratic particle movement.
Damping	When enabled, strength is proportional to frequency. Tying these values together means the noise field can be scaled while maintaining the same behaviour, but at a different size.
Octaves	Specify how many layers of overlapping noise are combined to produce the final noise values. Using more layers gives richer, more interesting noise, but significantly adds to the performance cost.
Octave Multiplier	For each additional noise layer, reduce the strength by this proportion.
Octave Scale	For each additional noise layer, adjust the frequency by this multiplier.
Quality	Lower quality settings reduce the performance cost significantly, but also affect how interesting the noise looks. Use the lowest quality that gives you the desired behavior for maximum performance.
Remap	Remap the final noise values into a different range.
Remap Curve	The curve that describes how the final noise values are transformed. For example, you could use this to pick out the lower ranges of the noise field and ignore the higher ranges by creating a curve that starts high and ends at zero.

Property	Function
Position Amount	A multiplier to control how much the noise affects particle positions.
Rotation Amount	A multiplier to control how much the noise affects particle rotations, in degrees per second.
Size Amount	A multiplier to control how much the noise affects particle sizes.

Details

Adding noise to your particles is a simple and effective way to create interesting patterns and effects. For example, imagine how embers from a fire move around, or how smoke swirls as it moves. Strong, high frequency noise could be used to simulate the fire embers, while soft, low frequency noise would be better suited to modeling a smoke effect.

For maximum control over the noise, you can enable the Separate Axes option. This allows you to control the strength and remapping on each axis independently.

The noise algorithm used is based on a technique called Curl Noise, which internally uses multiple samples of Perlin Noise to create the final noise field.

The quality settings control how many unique noise samples are generated. When using Medium and Low, less samples of Perlin Noise are used, and those samples are re-used across multiple axes but combined in a way to try and hide the re-use. This means that the noise may look less dynamic and diverse when using lower quality settings. However, there is a significant performance benefit when using lower quality settings.

2017–09–05 Page amended with editorial review

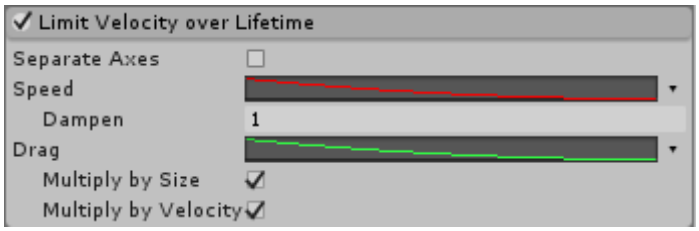
Position Amount, Rotation Amount, Size Amount added in Unity 2017.1

Strength, Frequency, noise algorithm and quality settings added in Unity 2017.2

Limit Velocity Over Lifetime module

Leave feedback

This module controls how the speed of particles is reduced over their lifetime.



Properties

Property	Function
Separate Axes	Splits the axes up into separate X, Y and Z **components**.
Speed	Sets the speed limit of the particles.
Space	Selects whether the speed limitation refers to local or world space. This option is only available when **Separate Axes** is enabled.
Dampen	The fraction by which a particle's speed is reduced when it exceeds the speed limit.
Drag	Applies linear drag to the particle velocities.
Multiply by Size	When enabled, larger particles are affected more by the drag coefficient.
Multiply by Velocity	When enabled, faster particles are affected more by the drag coefficient.

Details

This module is very useful for simulating air resistance that slows the particles, especially when a decreasing curve is used to lower the speed limit over time. For example, an explosion or firework initially bursts at great speed but the particles emitted from it rapidly slow down as they pass through the air.

The **Drag** option offers a more physically accurate simulation of air resistance by offering options to apply varying amounts of resistance based on the size and speed of the particles.

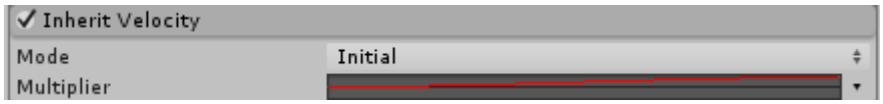
2017–09–05 Page amended with editorial review

Drag, Multiple by Size, Multiply by Velocity added in Unity 2017.2

Inherit Velocity module

Leave feedback

This module controls how the speed of particles reacts to movement of their parent object over time.



Properties

Property	Function
Mode	Specifies how the emitter velocity is applied to particles
Current	The emitter's current velocity will be applied to all particles on every frame. For example, if the emitter slows down, all particles will also slow down.
Initial	The emitter's velocity will be applied once, when each particle is born. Any changes to the emitter's velocity made after a particle is born will not affect that particle.
Multiplier	The proportion of the emitter's velocity that the particle should inherit.

Details

This effect is very useful for emitting particles from a moving object, such as dust clouds from a car, smoke from a rocket, steam from a steam train's chimney, or any situation where the particles should initially be moving at a percentage of the speed of the object they appear to come from. This module only has an effect on the particles when **Simulation Space** is set to **World** in the Main module.

It is also possible to use curves to influence the effect over time. For example, you could apply a strong attraction to newly created particles, which reduces over time. This could be useful for steam train smoke, which would drift off slowly over time and stop following the train it was emitted from.

Force Over Lifetime module

Leave feedback

Particles can be accelerated by forces (such as wind or attraction) that are specified in this module.

Properties

Property	*Function*
X, Y, Z	Force applied to each particle in the X, Y and Z axes.
Space	Selects whether the force is applied in local or world space.
Randomize	When using the Two Constants or Two Curves modes, this causes a new force direction to be chosen on each frame within the defined ranges. This causes more turbulent, erratic movement.

Details

Fluids are often affected by forces as they move. For example, smoke will accelerate slightly as it rises from a fire, carried up by the hot air around it. Subtle effects can be achieved by using curves to control the force over the particles' lifetimes. Using the previous example, smoke will initially accelerate upward but as the rising air gradually cools, the force will diminish. Thick smoke from a fire might initially accelerate, then slow down as it spreads and perhaps even start to fall to earth if it persists for a long time.

Color Over Lifetime module

Leave feedback

This module specifies how a particle's color and transparency changes over its lifetime.

Properties

Property:	*Function:*
Color	The color gradient of a particle over its lifetime. The very left-hand point of the gradient bar indicates the beginning of the particle's life, and the very right-hand side of the gradient bar indicates the end of the particle's life. In the image above, the particle starts off orange, fades in opacity over time, and is invisible by the time its life ends.

Details

Many types of natural and fantastical particles vary in color over time, and so this property has many uses. For example, white hot sparks will cool as they pass through the air and a magic spell might burst into a rainbow of colors. Equally important, though, is the variation of alpha (transparency). It is very common for particles to burn out, fade or dissipate as they reach the end of their lifetime (for example, hot sparks, fireworks and smoke particles) and a simple diminishing gradient produces this effect.

Color By Speed module

Leave feedback

The color of a particle can be set here to change according to its speed in distance units per second.

Properties

Property	*Function*
Color	The color gradient of a particle defined over a speed range.
Speed Range	The low and high ends of the speed range to which the color gradient is mapped (speeds outside the range will map to the end points of the gradient).

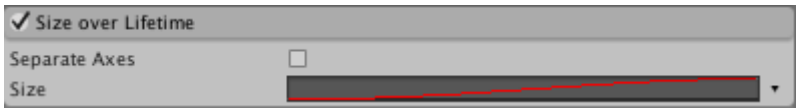
Details

Burning or glowing particles (such as sparks) tend to burn more brightly when they move quickly through the air (for example, when sparks are exposed to more oxygen), but then dim slightly as they slow down. To simulate this, you might use *Color By Speed* with a gradient that has white at the upper end of the speed range, and red at the lower end (in the spark example, faster particles will appear white while slower particles are red).

Size over Lifetime module

Leave feedback

Many effects involve a particle changing size according to a curve, which can be set in this module.



Properties

Property	*Function*
Separate Axes	Control the particle size independently on each axis.
Size	A curve which defines how the particle's size changes over its lifetime.

Details

Some particles will typically change in size as they move away from the point of emission, such as those that represent gases, flames or smoke. For example, smoke will tend to disperse and occupy a larger volume over time. You can achieve this by setting the curve for the smoke particle to an upward ramp, increasing with the particle's age. You can also further enhance this effect using the **Color Over Lifetime** module to fade the smoke as it spreads.

For fireballs created by burning fuel, the flame particles will tend to expand after emission but then fade and shrink as the fuel is used up and the flame dissipates. In this case, the curve would have a rising "hump" that rises and then falls back down to a smaller size.

Non-uniform particle scaling

You can specify how a particle's width, height and depth changes over lifetime independently. In the **Size over Lifetime** module, check the **Separate Axes** checkbox, then change the X (width), Y (height) and Z (depth). Remember that Z will only be used for **Mesh** particles.

Size by Speed module

Leave feedback

This module allows you to create particles that change size according to their speed in distance units per second.

Properties

Property	*Function*
Separate Axes	Control the particle size independently on each axis.
Size	A curve defining the particle's size over a speed range.
Speed Range	The low and high ends of the speed range to which the size curve is mapped (speeds outside the range will map to the end points of the curve).

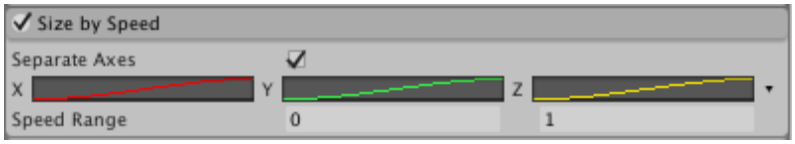
Details

Some situations will require particles which vary in size depending on their speed. For example, you would expect small pieces of debris to be accelerated more by an explosion than larger pieces. You can achieve effects like this using **Size By Speed** with a simple ramp curve that proportionally increases the speed as the size of the particle decreases. Note that this should not be used with the **Limit Velocity Over Lifetime** module, unless you want particles to change their size as they slow down.

Speed Range specifies the range of values that the X (width), Y (height) and Z (depth) shapes apply to. The Speed Range is only applied when the size is in one of the curve modes. Fast particles will scale using the values at the right end of the curve, while slower particles will use values from the left side of the curve. For example, if you specify a Speed Range between 10 and 100:

Speeds below 10 will set the Particle size corresponding with the left-most edge of the curve.
Speeds above 100 will set the Particle size corresponding with the right-most edge of the curve.
Speeds between 10 and 100 will set the Particle size determined by the point along the curve corresponding to the Speed. In this example, a Speed of 55 would set the size according to the midpoint of the curve.

Non-uniform particle scaling

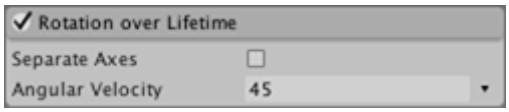


You can specify how a particle's width, height and depth size changes by speed independently. In the **Size by Speed** module, check the **Separate Axes** checkbox, then choose how the X (width), Y (height) and Z (depth) of the particle is affected by the speed of the particle. Remember that Z will only be used for **Mesh** particles.

Rotation Over Lifetime module

Leave feedback

Here, you can configure particles to rotate as they move.



Properties

Property	Function
Separate Axes	Allow rotation to be specified per axis. When this is enabled, the option to set a rotation for each of X, Y and Z is presented.
Angular Velocity	Rotation velocity in degrees per second. See below for more information.

Details

This setting is useful when particles represent small solid objects, such as pieces of debris from an explosion. Assigning random values of rotation will make the effect more realistic than having the particles remain upright as they fly. The random rotations will also help to break up the regularity of similarly shaped particles (the same texture repeated many times can be very noticeable).

Leaves rendered using particles with random 3D rotation

Options

The angular velocity option can be changed from the default constant speed. The drop-down to the right of the velocity can provide:

Property	Function
Constant	The velocity for particle rotation in degrees per second.
Curve	The angular velocity can be set to change over the lifetime of the particle. A curve editor appears at the bottom of the **Inspector** which allows you to control how the velocity changes throughout the lifetime of the particle (see Image A below). If the Separate Axes box is ticked, each of the X, Y and Z axes can be given curved velocity values.
Random Between Two Constants	The angular velocity properties has two angles allowing a rotation between them.

Property	Function
Random Between Two Curves	The angular velocity can be set to change over the lifetime of the particle specified by a curve. In this mode, two curves are editable, and each particle will pick a random curve between the range of these two curves that you define (see Image B below).

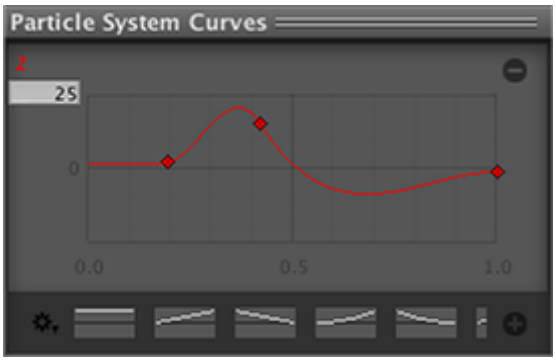


Image A: Z-axis angular velocity

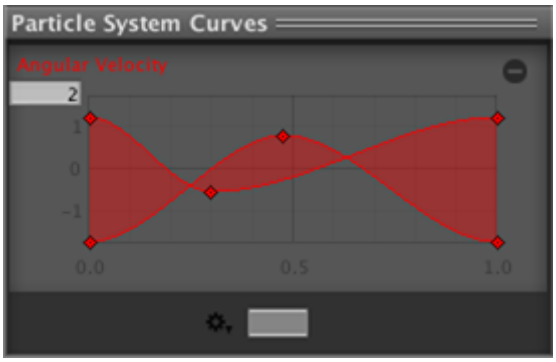


Image B: Angular velocity between two curves

Rotation By Speed module

Leave feedback

The rotation of a particle can be set here to change according to its speed in distance units per second.

Properties

Property	*Function*
Separate Axes	Control rotation independently for each axis of rotation.
Angular Velocity	Rotation velocity in degrees per second.
Speed Range	The low and high ends of the speed range to which the size curve is mapped (speeds outside the range will map to the end points of the curve).

Details

This property can be used when the particles represent solid objects moving over the ground such as rocks from a landslide. The rotation of the particles can be set in proportion to the speed so that they roll over the surface convincingly.

The Speed Range is only applied when the velocity is in one of the curve modes. Fast particles will rotate using the values at the right end of the curve, while slower particles will use values from the left side of the curve.

External Forces module

Leave feedback

This property modifies the effect of **wind zones** on particles emitted by the system.

Properties

Property	*Function*
Multiplier	Scale value applied to wind zone forces.

Details

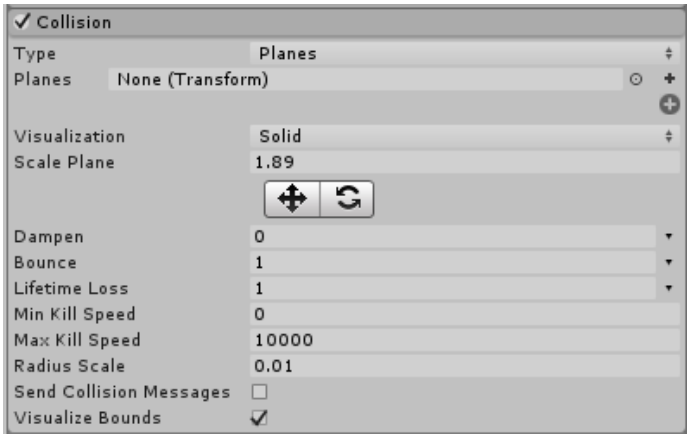
A **Terrain** can incorporate *wind zones* which affect the movement of trees on the landscape. Enabling this section allows the wind zones to blow particles from the system. The *Multiplier* value lets you scale the effect of the wind on the particles, since they will often be blown more strongly than tree branches.

Collision module

Leave feedback

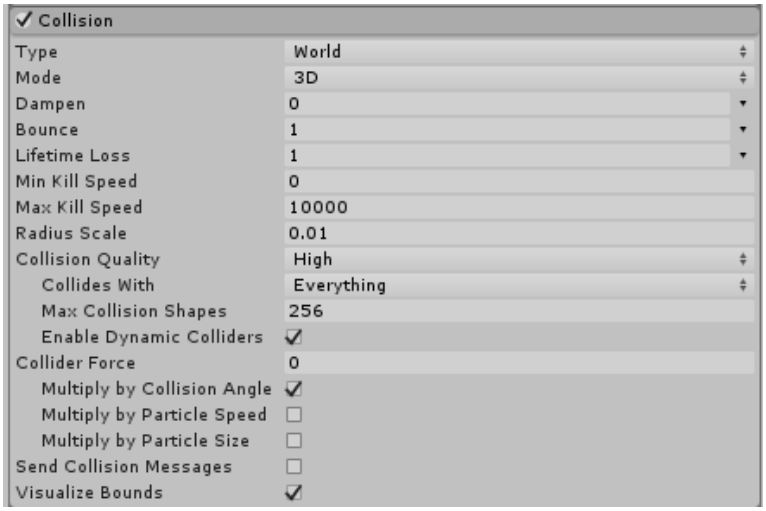
This module controls how particles collide with GameObjects in the Scene. Use the first drop-down to define whether your collision settings apply to **Planes** or to the **World**. If you choose **World**, use the **Collision Mode** drop-down to define whether your collision settings apply for a 2D or 3D world.

Planes module properties



Property	Function
Planes popup	Select **Planes** mode.
Planes	An expandable list of Transforms that define collision planes.
Visualization	Selects whether the collision plane **Gizmos** will be shown in the **Scene view** as wireframe grids or solid planes.
Scale Plane	Size of planes used for visualization.
Dampen	The fraction of a particle's speed that it loses after a collision.
Bounce	The fraction of a particle's speed that rebounds from a surface after a collision.
Lifetime Loss	The fraction of a particle's total lifetime that it loses if it collides.
Min Kill Speed	Particles travelling below this speed after a collision will be removed from the system.
Max Kill Speed	Particles travelling above this speed after a collision will be removed from the system.
Radius Scale	Allows you to adjust the radius of the particle collision spheres so it more closely fits the visual edges of the particle graphic.
Send Collision Messages	If enabled, particle collisions can be detected from **scripts** by the OnParticleCollision function.
Visualize Bounds	Renders the collision bounds of each particle as a wireframe shape in the Scene view.

World module properties



Property	*Function*

Property	*Function*
World popup	Select **World** mode.
Collision Mode	3D or 2D.
Dampen	The fraction of a particle's speed that it loses after a collision.
Bounce	The fraction of a particle's speed that rebounds from a surface after a collision.
Lifetime Loss	The fraction of a particle's total lifetime that it loses if it collides.
Min Kill Speed	Particles travelling below this speed after a collision will be removed from the system.
Max Kill Speed	Particles travelling above this speed after a collision will be removed from the system.
Radius Scale	Setting for 2D or 3D.
Collision Quality	Use the drop-down to set the quality of particle collisions. This affects how many particles can pass through a collider. At lower quality levels, particles can sometimes pass through colliders, but are less resource-intensive to calculate.
High	When **Collision Quality** is set to **High**, collisions always use the physics system for detecting the collision results. This is the most resource-intensive option, but also the most accurate.
Medium (Static Colliders)	When **Collision Quality** is set to **Medium (Static Colliders)**, collisions use a grid of voxels to cache previous collisions, for faster re-use in later frames. See World collisions, below, to learn more about this cache. The only difference between **Medium** and **Low** is how many times per frame the Particle System queries the physics system. **Medium** makes more queries per frame than **Low**. Note that this setting is only suitable for static colliders that never move.
Low (Static Colliders)	When **Collision Quality** is set to **Low (Static Colliders)**, collisions use a grid of voxels to cache previous collisions, for faster re-use in later frames. See World collisions, below, to learn more about this cache. The only difference between **Medium** and **Low** is how many times per frame the Particle System queries the physics system. **Medium** makes more queries per frame than **Low**. Note that this setting is only suitable for static colliders that never move.
Collides With	Particles will only collide with objects on the selected layers.
Max Collision Shapes	How many collision shapes can be considered for particle collisions. Excess shapes are ignored, and **terrains** take priority.
Enable Dynamic Colliders	Allows the particles to also collide with dynamic objects (otherwise only static objects are used).
Enable Dynamic Colliders	Dynamic colliders are any collider not configured as Kinematic (see documentation on colliders for further information on collider types). Check this option to include these collider types in the set of objects that the particles respond to in collisions. Uncheck this option and the particles only respond to collisions against static colliders.
Voxel Size	A voxel represents a value on a regular grid in three-dimensional space. When using **Medium** or **Low** quality collisions, Unity caches collisions in a grid structure. This setting controls the grid size. Smaller values give more accuracy, but cost more memory, and are less efficient. **Note**: You can only access this property when **Collision Quality** is set to **Medium** or **Low**.

Property	*Function*
Collider Force	Apply a force to Physics Colliders after a Particle collision. This is useful for pushing colliders with particles.
Multiply by Collision Angle	When applying forces to Colliders, scale the strength of the force based on the collision angle between the particle and the collider. Grazing angles will generate less force than a head-on collision.
Multiply by Particle Speed	When applying forces to Colliders, scale the strength of the force based on the speed of the particle. Fast-moving particles will generate more force than slower ones.
Multiply by Particle Size	When applying forces to Colliders, scale the strength of the force based on the size of the particle. Larger particles will generate more force than smaller ones.
Send Collision Messages	Check this to be able to detect particle collisions from scripts by the OnParticleCollision function.
Visualize Bounds	Preview the collision spheres for each particle in the Scene view.

Details

When other objects surround a **Particle System**, the effect is often more convincing when the particles interact with those objects. For example, water or debris should be obstructed by a solid wall rather than simply passing through it. With the **Collision** module enabled, particles can collide with objects in the Scene.

A Particle System can be set so its particles collide with any Collider in the scene by selecting **World** mode from the pop-up. Colliders can also be disabled according to the layer they are on by using the **Collides With** property. The pop-up also has a **Planes** mode option which allows you to add a set of planes to the Scene that don't need to have Colliders. This option is useful for simple floors, walls and similar objects, and has a lower processor overhead than **World** mode.

When **Planes** mode is enabled, a list of transforms (typically empty GameObjects) can be added via the **Planes** property. The planes extend infinitely in the objects' local XZ planes, with the positive Y axis indicating the planes' normal vectors. To assist with development, the planes will be shown as Gizmos in the Scene, regardless of whether or not the objects have any visible **Mesh** themselves. The Gizmos can be shown as a wireframe grid or a solid plane, and can also be scaled. However, the scaling only applies to the visualization - the collision planes themselves extend infinitely through the Scene.

When collisions are enabled, the size of a particle is sometimes a problem because its graphic can be clipped as it makes contact with a surface. This can result in a particle appearing to "sink" partway into a surface before stopping or bouncing. The **Radius Scale** property addresses this issue by defining an approximate circular radius for the particles, as a percentage of its actual size. This size information is used to prevent clipping and avoid the sinking-in effect.

The **Dampen** and **Bounce** properties are useful for when the particles represent solid objects. For example, gravel will tend to bounce off a hard surface when thrown but a snowball's particles might lose speed during a collision. **Lifetime Loss** and **Min Kill Speed** can help to reduce the effects of residual particles following a collision. For example, a fireball might last for a few seconds while flying through the air but after colliding, the separate fire particles should dissipate quickly.

You can also detect particle collisions from a script if **Send Collision Messages** is enabled. The script can be attached to the object with the particle system, or the one with the Collider, or both. By detecting collisions, you can use particles as active objects in gameplay, for example as projectiles, magic spells and power-ups. See the script reference page for MonoBehaviour.OnParticleCollision for further details and examples.

World Collision Quality

The World Collision module has a **Collision Quality** property, which you can set to **High**, **Medium** or **Low**. When **Collision Quality** is set to **Medium (Static Colliders)** or **Low (Static Colliders)**, collisions use a grid of **voxels** (values on a 3D grid) to cache previous collisions, for fast re-use in later frames.

This cache consists of a plane in each voxel, where the plane represents the collision surface at that location. On each frame, Unity checks the cache for a plane at the position of the particle, and if there is one, Unity uses it for **collision detection**. Otherwise, it asks the physics system. If a collision is returned, it is added to the cache for fast querying on subsequent frames.

This is an approximation, so some missed collisions might occur. You can reduce the Voxel Size value to help with this; however, doing so uses extra memory, and is less efficient.

The only difference between **Medium** and **Low** is how many times per frame the system is allowed to query the physics system. Low makes fewer queries per frame than Medium. Once the per-frame budget has been exceeded, only the cache is used for any remaining particles. This can lead to an increase in missed collisions, until the cache has been more comprehensively populated.

2017–05–30 Page amended with limited editorial review

Functionality of Collision module changed in Unity 2017.1

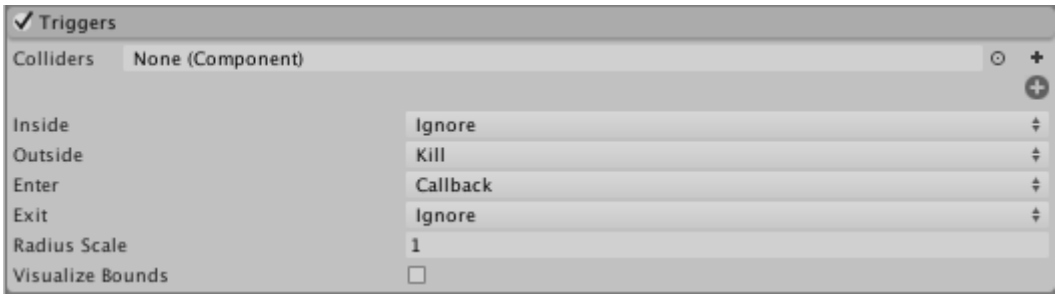
Triggers module

Leave feedback

Particle Systems have the ability to trigger a Callback whenever they interact with one or more Colliders in the **Scene**. The Callback can be triggered when a particle enters or exits a **Collider**, or during the time that a particle is inside or outside of the **Collider**.

You can use the Callback as a simple way to destroy a particle when it enters the Collider (for example, to prevent raindrops from penetrating a rooftop), or it can be used to modify any or all particles' properties.

The Triggers module also offers the **Kill** option to remove particles automatically, and the **Ignore** option to ignore **collision** events, shown below.



Particle Systems Triggers module

To use the module, first add the Colliders you wish to create triggers for, then select which events to use.

You can elect to trigger an event whenever the particle is:

Inside a Collider's bounds
Outside a Collider's bounds
Entering a Collider's bounds
Exiting a Collider's bounds

Property:	*Function:*
Inside	Select Callback if you want the event to trigger when the particle is inside the Collider. Select Ignore for the event **not** to trigger when the particle is inside the Collider. Select Kill to destroy particles inside the Collider.
Outside	Select Callback if you want the event to trigger when the particle is outside the Collider. Select Ignore for the event **not** to trigger when the particle is outside the Collider. Select Kill to destroy particles outside the Collider.
Enter	Select Callback if you want the event to trigger when the particle enters the Collider. Select Ignore for the event **not** to trigger when the particle enters the Collider. Select Kill to destroy particles when they enter the Collider.
Exit	Select Callback if you want the event to trigger when the particle exits the Collider. Select Ignore for the event **not** to trigger when the particle exits the Collider. Select Kill to destroy particles when they exit the Collider.

Property:	*Function:*
Radius Scale	This parameter sets the particle's Collider bounds, allowing an event to appear to happen before or after the particle touches the Collider. For example, you may want a particle to appear to penetrate a Collider object's surface a little before bouncing off, in which case you would set the Radius Scale to be a little less than 1. Note that this setting does not change when the event actually triggers, but can delay or advance the visual effect of the trigger. - Enter 1 for the event to appear to happen as a Particle touches the Collider - Enter a value less than 1 for the trigger to appear to happen after a Particle penetrates the Collider - Enter a value greater than 1 for the trigger to appear to happen after a Particle penetrates the Collider
Visualize Bounds	This allows you to display the Particle's Collider bounds in the Editor window.

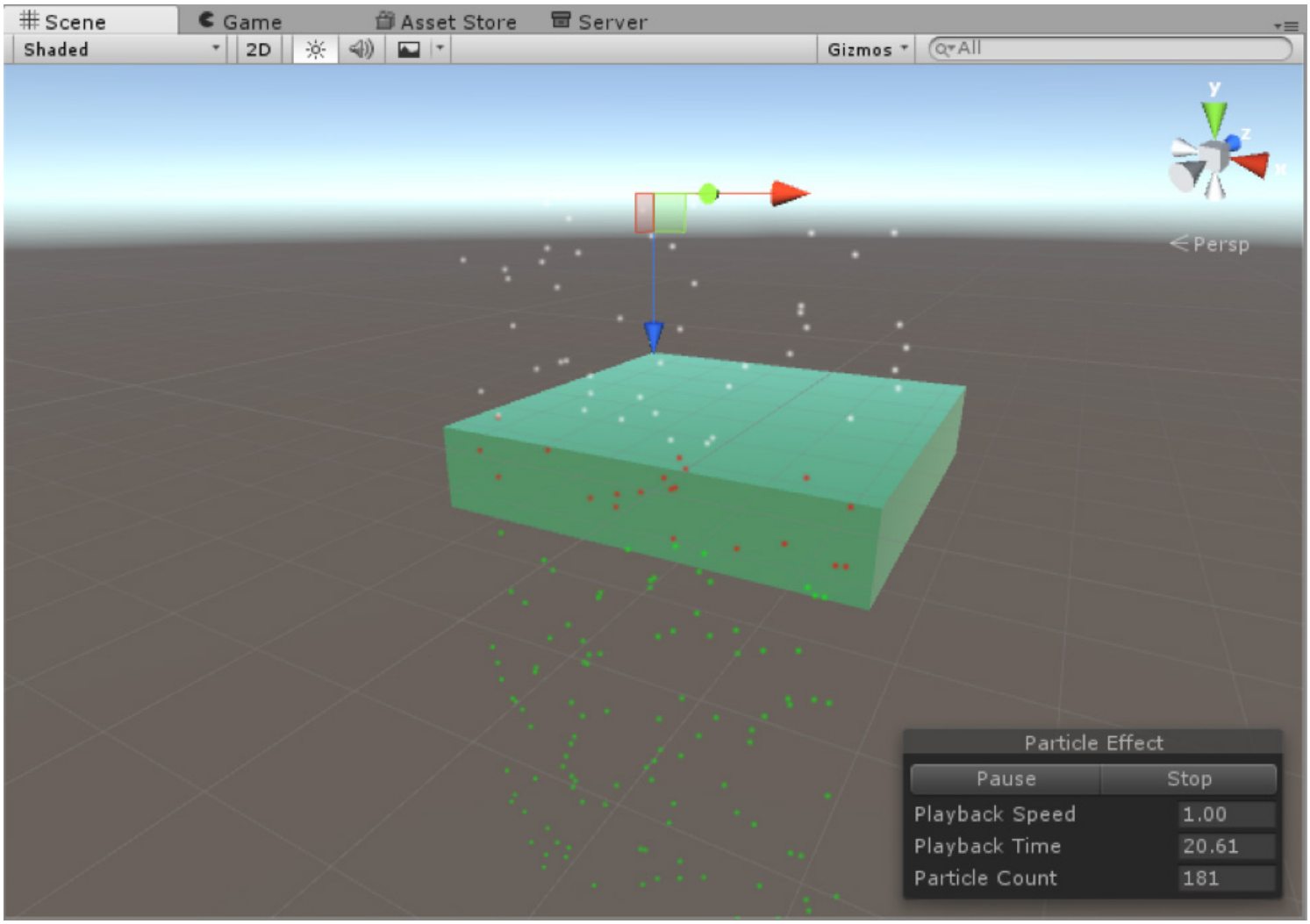
Inside the Callback, use ParticlePhysicsExtensions.GetTriggerParticles() (along with the ParticleSystemTriggerEventType you want to specify) to determine which particles meet which criteria.

The example below causes particles to turn red as they enter the Collider, then turn green as they leave the Collider's bounds.

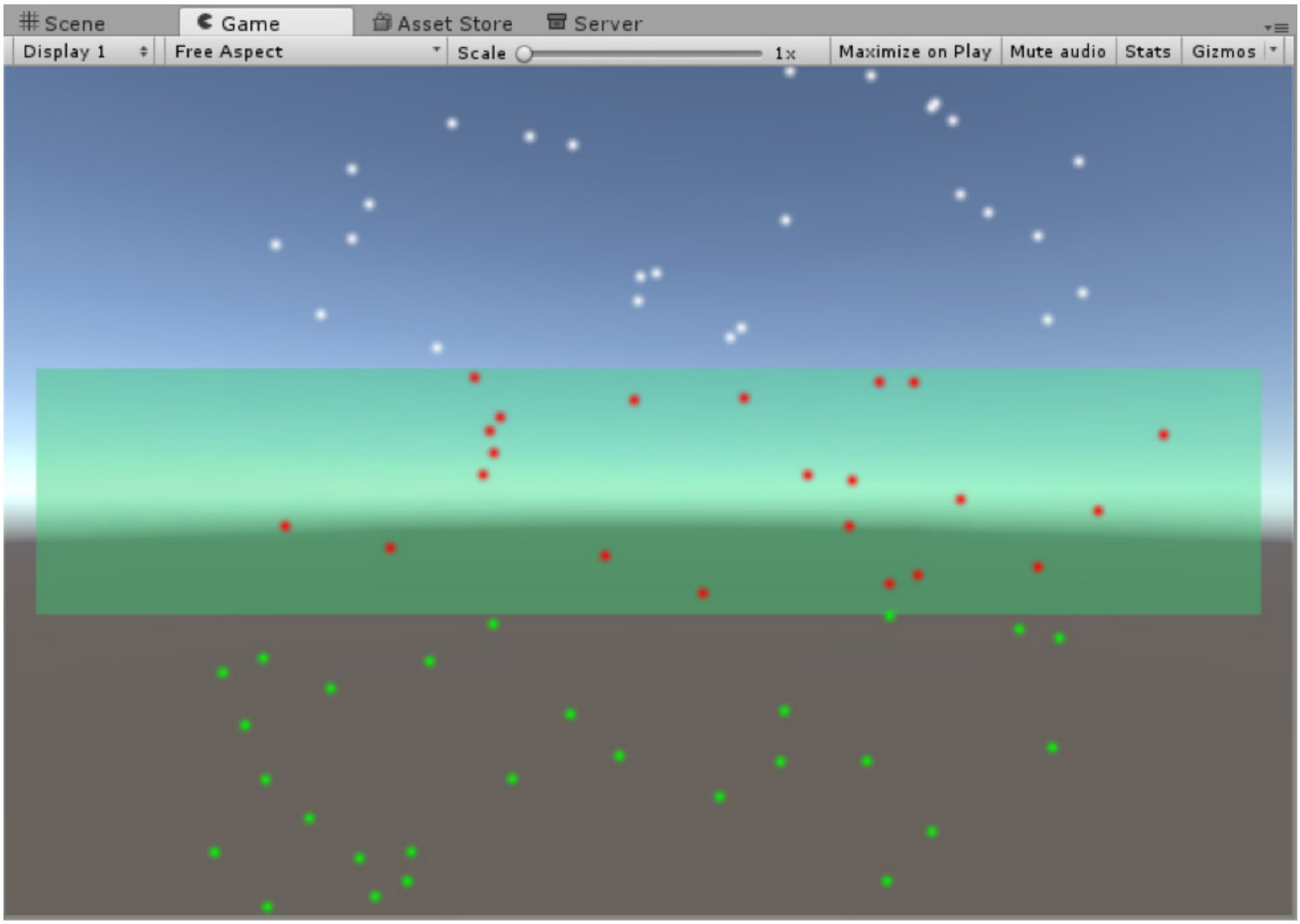
```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[ExecuteInEditMode]
public class TriggerScript : MonoBehaviour
{
    ParticleSystem ps;

    // these lists are used to contain the particles which match
    // the trigger conditions each frame.
    List<ParticleSystem.Particle> enter = new List<ParticleSystem.Particle>();
    List<ParticleSystem.Particle> exit = new List<ParticleSystem.Particle>();

    void OnEnable()
    {
        ps = GetComponent<ParticleSystem>();
    }

    void OnParticleTrigger()
    {
        // get the particles which matched the trigger conditions this frame
        int numEnter = ps.GetTriggerParticles(ParticleSystemTriggerEventType.Ent
        int numExit = ps.GetTriggerParticles(ParticleSystemTriggerEventType.Exit
```

```
        // iterate through the particles which entered the trigger and make them
        for (int i = 0; i < numEnter; i++)
        {
            ParticleSystem.Particle p = enter[i];
            p.startColor = new Color32(255, 0, 0, 255);
            enter[i] = p;
        }

        // iterate through the particles which exited the trigger and make them
        for (int i = 0; i < numExit; i++)
        {
            ParticleSystem.Particle p = exit[i];
            p.startColor = new Color32(0, 255, 0, 255);
            exit[i] = p;
        }

        // re-assign the modified particles back into the particle system
        ps.SetTriggerParticles(ParticleSystemTriggerEventType.Enter, enter);
        ps.SetTriggerParticles(ParticleSystemTriggerEventType.Exit, exit);
    }
}
```

The result of this is demonstrated in the images below:


Scene
Game
Asset Store
Server
Shaded
2D
Gizmos
All
y
z
x
Persp
Particle Effect
Pause
Stop
Playback Speed
1.00
Playback Time
20.61
Particle Count
181

Editor view

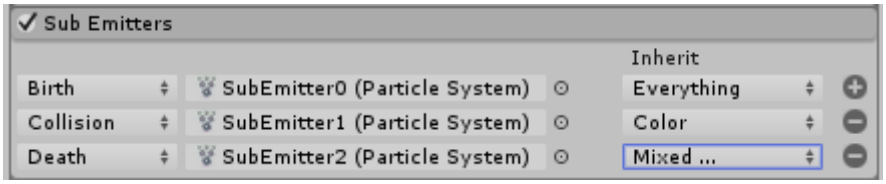

Scene
Game
Asset Store
Server
Display 1
Free Aspect
Scale
1x
Maximize on Play
Mute audio
Stats
Gizmos

Game view

Sub Emitters module

Leave feedback

This module allows you to set up sub-emitters. These are additional particle emitters that are created at a particle's position at certain stages of its lifetime.



Properties

Property	*Function*
Sub Emitters	Configure a list of sub-emitters and select their trigger condition as well as what properties they inherit from their parent particles.

Details

Many types of particles produce effects at different stages of their lifetimes that can also be implemented using **Particle Systems**. For example, a bullet might be accompanied by a puff of smoke powder as it leaves the gun barrel, and a fireball might explode on impact. You can use sub-emitters to create effects like these.

Sub-emitters are ordinary Particle System objects created in the **Scene** or from Prefabs. This means that sub-emitters can have sub-emitters of their own (this type of arrangement can be useful for complex effects like fireworks). However, it is very easy to generate an enormous number of particles using sub-emitters, which can be resource intensive.

To trigger a sub-emitter, you can use these are the conditions:

Birth: When the particles are created.
Collision: When the particles collide with an object.
Death: When the particles are destroyed.
Trigger: When the particles interact with a Trigger **collider**.
Manual: Only triggered when requested via script. See ParticleSystem.TriggerSubEmitter.
Note that the **Collision**, **Trigger**, **Death** and **Manual** events can only use burst emission in the Emission module.

Additionally, you can transfer properties from the parent particle to each newly created particle using the **Inherit** options. The transferrable properties are size, rotation, color and lifetime. To control how velocity is inherited, configure the Inherit Velocity module on the sub-emitter system.

2018–03–28 Page amended with limited editorial review

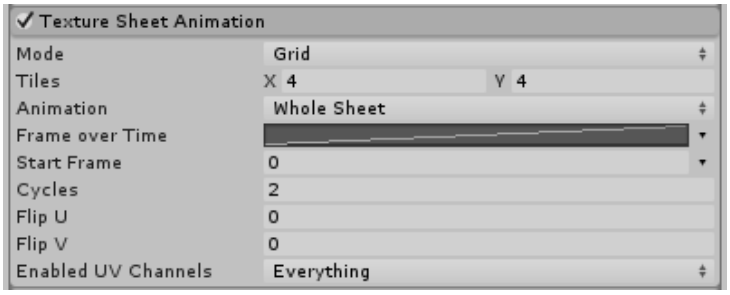
"Trigger" and "Manual" conditions added to conditions list in Sub Emitters Module in 2018.1

Texture Sheet Animation module

Leave feedback

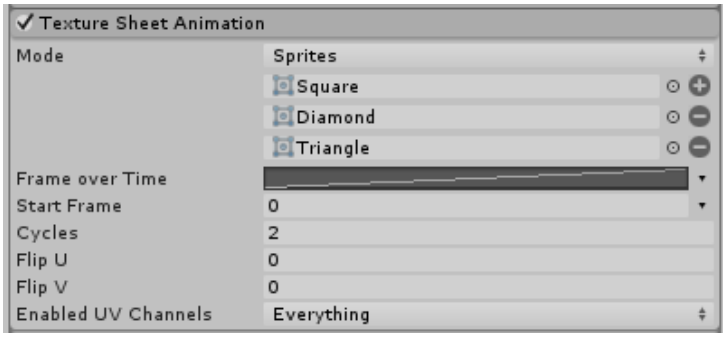
A particle's graphic need not be a still image. This module lets you treat the Texture as a grid of separate sub-images that can be played back as frames of animation.

Grid mode properties



Property	Function
Mode popup	Select **Grid** mode.
Tiles	The number of tiles the Texture is divided into in the X (horizontal) and Y (vertical) directions.
Animation	The Animation mode can be set to **Whole Sheet** or **Single Row** (that is, each row of the sheet represents a separate Animation sequence).
Random Row	Chooses a row from the sheet at random to produce the animation. This option is only available when **Single Row** is selected as the **Animation** mode.
Row	Selects a particular row from the sheet to produce the animation This option is only available when **Single Row** mode is selected and **Random Row** is disabled.
Frame over Time	A curve that specifies how the frame of animation increases as time progresses.
Start Frame	Allows you to specify which frame the particle animation should start on (useful for randomly phasing the animation on each particle).
Cycles	The number of times the animation sequence repeats over the particle's lifetime.
Flip U	Horizontally mirror the Texture on a proportion of the particles. A higher value flips more particles.
Flip V	Vertically mirror the Texture on a proportion of the particles. A higher value flips more particles.
Enabled UV Channels	Allows you to specify exactly which UV streams are affected by the **Particle System**.

Sprite mode properties



Property	Function
Mode popup	Select **Sprites** mode.
Frame over Time	A curve that specifies how the frame of animation increases as time progresses.
Start Frame	Allows you to specify which frame the particle animation should start on (useful for randomly phasing the animation on each particle).

Property	Function
Cycles	The number of times the animation sequence repeats over the particle's lifetime.
Flip U	Horizontally mirror the Texture on a proportion of the particles. A higher value flips more particles.
Flip V	Vertically mirror the Texture on a proportion of the particles. A higher value flips more particles.
Enabled UV Channels	Allows you to specify exactly which UV streams are affected by the Particle System.

Details

Particle animations are typically simpler and less detailed than character animations. In systems where the particles are visible individually, animations can be used to convey actions or movements. For example, flames may flicker and insects in a swarm might vibrate or shudder as if flapping their wings. In cases where the particles form a single, continuous entity like a cloud, animated particles can help add to the impression of energy and movement.

You can use the **Single Row** mode to create separate animation sequences for particles and switch between animations from a script. This can be useful for creating variation or switching to a different animation after a **collision**. The **Random Row** option is highly effective as a way to break up conspicuous regularity in a particle system (for example, a group of flame objects that are all repeating the exact same flickering animation over and over again). This option can also be used with a single frame per row as a way to generate particles with random graphics. This can be used to break up regularity in a object like a cloud or to produce different types of debris or other objects from a single system. For example, a blunderbuss might fire out a cluster of nails, bolts, balls and other projectiles, or a car crash effect may result in springs, car paint, screws and other bits of metal being emitted.

UV flipping is a great way to add more visual variety to your effects without needing to author additional textures.

Selecting the **Sprites** option from the Mode dropdown allows you to define a list of Sprites to be displayed for each particle, instead of using a regular grid of frames on a texture. Using this mode allows you to take advantage of many of the features of Sprites, such as the **Sprite Packer**, custom pivots and different sizes per Sprite frame. The **Sprite Packer** can help you share materials between different Particle Systems, by atlasing your textures, which in turn can improve performance via **Dynamic Batching**. There are some limitations to be aware of with this mode. Most importantly, all Sprites attached to a Particle System must share the same texture. This can be achieved by using a Multiple Mode Sprite, or by using the **Sprite Packer**. If using custom pivot points for each Sprite, please note that you cannot blend between their frames, because the geometry will be different between each frame. Only simple Sprites are supported, not 9 slice. Also be aware that **Mesh** particles do not support custom pivots or varying Sprite sizes.

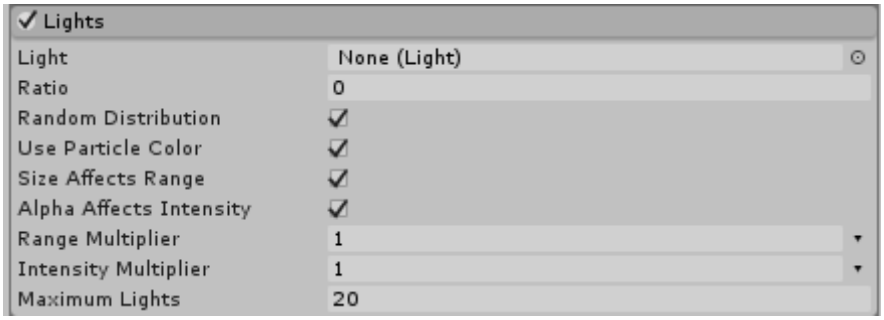
2017–05–30 Page amended with limited editorial review

Functionality of Texture Sheet Animation module changed in Unity 2017.1

Lights module

Leave feedback

Add **real-time lights** to a percentage of your particles using this module.



Properties

Property	Function
Light	Assign a Light **Prefab** describing how your particle lights should look.
Ratio	A value between 0 and 1 describing the proportion of particles that will receive a light.
Random Distribution	Choose whether lights are assigned randomly or periodically. When set to true, every particle has a random chance of receiving a light based on the Ratio. Higher values increase the probability of a particle having a light. When set to false, the Ratio controls how often a newly created particle receives a light (for example, every Nth particle will receive a light).
Use Particle Color	When set to True, the final color of the Light will be modulated by the color of the particle it is attached to. If set to False, the Light color is used without any modification.
Size Affects Range	When enabled, the **Range** specified in the Light will be multiplied by the size of the particle.
Alpha Affects Intensity	When enabled, the **Intensity** of the light is multiplied by the particle alpha value.
Range Multiplier	Apply a custom multiplier to the Range of the light over the lifetime of the particle using this curve.
Intensity Multiplier	Apply a custom multiplier to the Intensity of the light over the lifetime of the particle using this curve.
Maximum Lights	Use this setting to avoid accidentally creating an enormous number of lights, which could make the Editor unresponsive or make your application run very slowly.

Details

The Lights module is a fast way to add real-time lighting to your particle effects. It can be used to make systems cast light onto their surroundings, for example for a fire, fireworks or lightning. It also allows you to have the lights inherit a variety of properties from the particles they are attached to. This can make it more believable that the particle effect itself is emitting light. For example, this can be achieved by making the lights fade out with their particles and having them share the same colors.

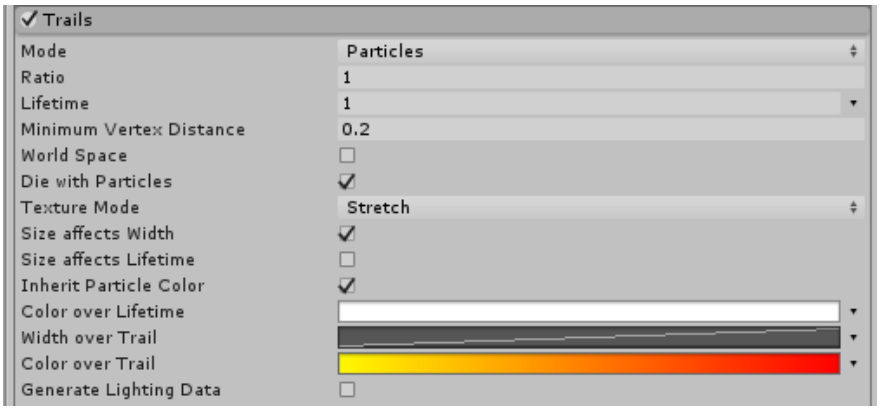
This module makes it easy to create a lot of real-time lights very quickly, and real-time lights have a high performance cost, especially in **Forward Rendering** mode. If the lights also cast shadows, the performance cost

is even higher. To help guard against an accidental tweak to the emission rate and thus causing thousands of real-time lights to be created, the Maximum Lights property should be used. Creating more lights than your target hardware is able to manage can cause slowdowns and unresponsiveness.

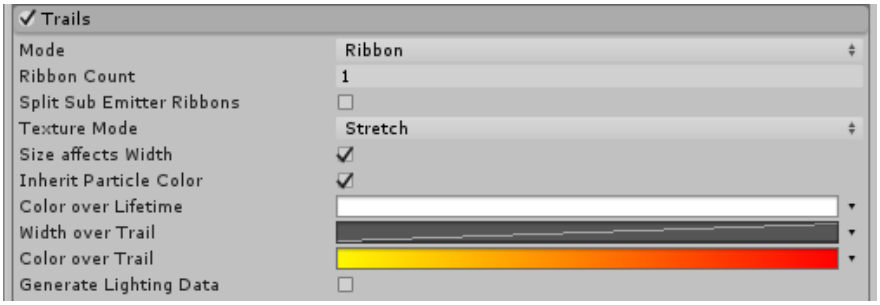
Trails module

Leave feedback

Add trails to a percentage of your particles using this module. This module shares a number of properties with the Trail Renderer **component**, but offers the ability to easily attach Trails to particles, and to inherit various properties from the particles. Trails can be useful for a variety of effects, such as bullets, smoke, and magic visuals.



The Trails module in Particles mode



The Trails module in Ribbon mode

Properties

Property:	Function:
Mode	Choose how trails are generated for the Particle System. - **Particle** mode creates an effect where each particle leaves a stationary trail in its path. - **Ribbon** mode create a ribbon of trails connecting each particle based on its age.
Ratio	A value between 0 and 1, describing the proportion of particles that have a Trail assigned to them. Unity assigns trails randomly, so this value represents a probability.
Lifetime	The lifetime of each vertex in the Trail, expressed as a multiplier of the lifetime of the particle it belongs to. When each new vertex is added to the Trail, it disappears after it has been in existence longer than its total lifetime.
Minimum Vertex Distance	Define the distance a particle must travel before its Trail receives a new vertex.
World Space	When enabled, Trail vertices do not move relative to the **Particle System**'s **GameObject**, even if using **Local Simulation Space**. Instead, the Trail vertices are dropped in the world, and ignore any movement of the **Particle System**.
Die With Particles	If this box is checked, Trails vanish instantly when their particles die. If this box is not checked, the remaining Trail expires naturally based on its own remaining lifetime.
Ribbon Count	Choose how many ribbons to render throughout the Particle System. A value of 1 creates a single ribbon connecting each particle. However, a value higher than 1 creates ribbons that connect every Nth particle. For example, when using a value of 2, there will be one ribbon connecting particles 1, 3, 5, and another ribbon connecting particles 2, 4, 6, and so on. The ordering of the particles is decided based on their age.

Property:	Function:
Split Sub Emitter Ribbons	When enabled on a system that is being used as a sub-emitter, particles that were spawned from the same parent system particle share a ribbon.
Texture Mode	Choose whether the Texture applied to the Trail is stretched along its entire length, or if it repeats every N units of distance. The repeat rate is controlled based on the **Tiling** parameters in the **Material**.
Size affects Width	If enabled (the box is checked), the Trail width is multiplied by the particle size.
Size affects Lifetime	If enabled (the box is checked), the Trail lifetime is multiplied by the particle size.
Inherit Particle Color	If enabled (the box is checked), the Trail color is modulated by the particle color.
Color over Lifetime	A curve to control the color of the entire Trail over the lifetime of the particle it is attached to.
Width over Trail	A curve to control the width of the Trail over its length.
Color over Trail	A curve to control the color of the Trail over its length.
Generate Lighting Data	Enable this (check the box), to build the Trail geometry with Normals and Tangents included. This allows them to use Materials that use the **scene** lighting, for example via the **Standard Shader**, or by using a custom **shader**.

Tips

Use the Renderer Module to specify the Trail Material.
Unity samples colors from the Color Gradient at each vertex, and linearly interpolates colors between each vertex,. Add more vertices to your **Line Renderer** to get a closer approximation of a detailed Color Gradient.

2017–10–26 Page amended with limited editorial review

Size affects Width, **Size affects Lifetime**, **Color over Lifetime**, **Width over Trail**, **Color over Trail**, and **Generate Lighting Data** added in Unity 2017.1

Particle mode added in Unity 2017.3

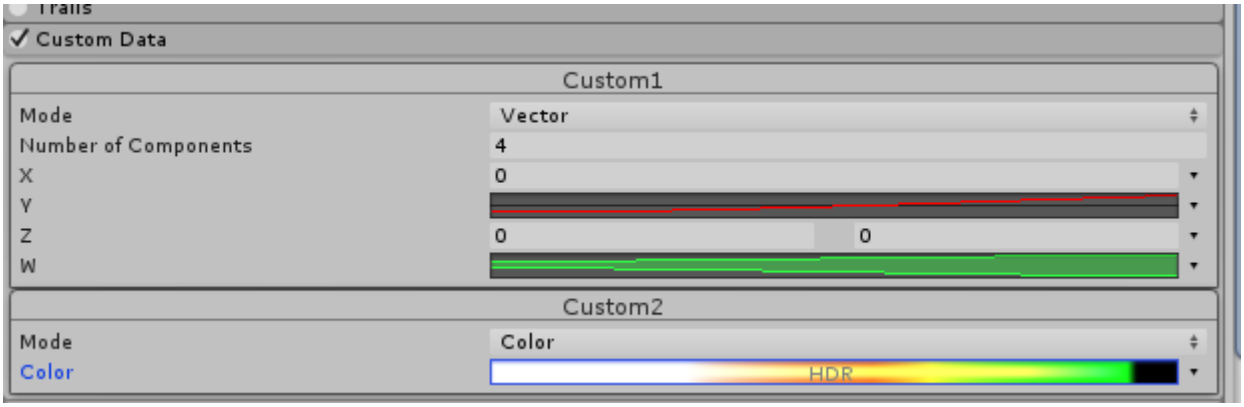
Custom Data module

Leave feedback

The Custom Data module allows you to define custom data formats in the Editor to be attached to particles. You can also set this in a script. See documentation on Particle System vertex streams for more information on how to set custom data from a script and feed that data into your **shaders**.

Data can be in the form of a **Vector**, with up to 4 MinMaxCurve **components**, or a **Color**, which is an HDR-enabled MinMaxGradient. Use this data to drive custom logic in your **scripts** and Shaders.

The default labels for each curve/gradient can be customized by clicking on them and typing in a contextual name. When passing custom data to shaders, it is useful to know how that data is used inside the shader. For example, a curve may be used for custom alpha testing, or a gradient may be used to add a secondary color to particles. By editing the labels, it is simple to keep a record in the **UI** of the purpose of each custom data entry.



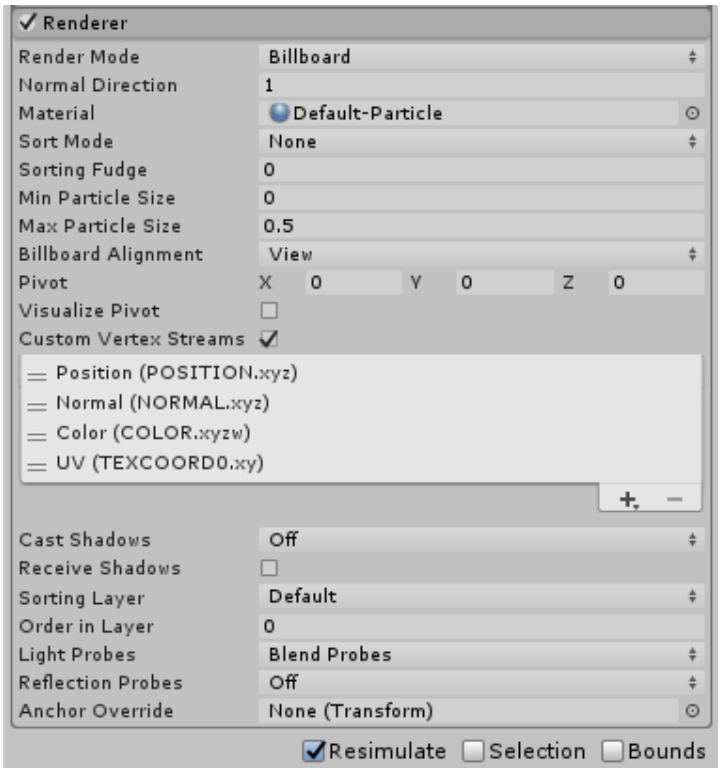
2017–09–04 Page amended with limited editorial review

Editable custom data labels added in Unity 2017.2

Renderer module

Leave feedback

The Renderer module's settings determine how a particle's image or Mesh is transformed, shaded and overdrawn by other particles.



Properties

Property	Function
Render Mode	How the rendered image is produced from the graphic image (or Mesh). See Details, below, for more information.
Billboard	The particle always faces the **Camera**.
Stretched Billboard	The particle faces the Camera but with various scaling applied (see below).
Camera Scale	Stretches particles according to Camera movement. Set this to 0 to disable Camera movement stretching.
Velocity Scale	Stretches particles proportionally to their speed. Set this to 0 to disable stretching based on speed.
Length Scale	Stretches particles proportionally to their current size along the direction of their velocity. Setting this to 0 makes the particles disappear, having effectively 0 length.
Horizontal Billboard	The particle plane is parallel to the XZ "floor" plane.
Vertical Billboard	The particle is upright on the world Y-axis, but turns to face the Camera.
Mesh	The particle is rendered from a 3D Mesh instead of a Texture.
None	This can be useful when using the Trails module, if you want to only render the trails and hide the default **rendering**.
Normal Direction	Bias of lighting normals used for the particle graphics. A value of 1.0 points the normals at the Camera, while a value of 0.0 points them towards the center of the screen (Billboard modes only).

Property	Function
Material	Material used to render the particles.
Trail Material	Material used to render particle trails. This option is only available when the **Trails** module is enabled.
Sort Mode	The order in which particles are drawn (and therefore overlaid). The possible values are **By Distance (from the Camera**), **Oldest in Front**, and **Youngest in Front**. Each particle within a system will be sorted according to this setting.
Sorting Fudge	Bias of particle system sort ordering. Lower values increase the relative chance that **particle systems** are drawn over other transparent **GameObjects**, including other **particle systems**. This setting only affects where an entire system appears in the **scene** - it does not perform sorting on individual particles within a system.
Min Particle Size	The smallest particle size (regardless of other settings), expressed as a fraction of **viewport** size. Note that this setting is only applied when the **Rendering Mode** is set to **Billboard**.
Max Particle Size	The largest particle size (regardless of other settings), expressed as a fraction of viewport size. Note that this setting is only applied when the **Rendering Mode** is set to **Billboard**.
Render Alignment	Use the drop-down to choose which direction particle billboards face.
View	Particles face the Camera plane.
World	Particles are aligned with the world axes.
Local	Particles are aligned to the Transform **component** of their GameObject.
Facing	Particles face the direct position of the Camera GameObject.
Enable GPU Instancing	Control whether the Particle System will be rendered using GPU Instancing. Requires using the Mesh Render Mode, and using a compatible **shader**. See Particle Mesh GPU Instancing for more details.
Pivot	Modify the pivot point used as the center for rotating particles.
Visualize Pivot	Preview the particle pivot points in the **Scene View**.
Custom Vertex Streams	Configure which particle properties are available in the **Vertex Shader** of your Material. See Particle Vertex Streams for more details.
Cast Shadows	If enabled, the particle system creates shadows when a shadow-casting Light shines on it.
On	Select **On** to enable shadows.
Off	Select **Off** to disable shadows.
Two-Sided	Select **Two Sided** to allow shadows to be cast from either side of the Mesh (meaning backface culling is not taken into account).
Shadows Only	Select **Shadows Only** to make it so that the shadows are visible, but the Mesh itself is not.
Receive Shadows	Dictates whether shadows can be cast onto particles. Only opaque materials can receive shadows.
Sorting Layer	Name of the Renderer's sorting layer.
Order in Layer	This Renderer's order within a sorting layer.

Property	Function
Light Probes	Probe-based lighting **interpolation** mode.
Reflection Probes	If enabled and reflection probes are present in the Scene, a reflection texture is picked for this GameObject and set as a built-in Shader uniform variable.
Anchor Override	A Transform used to determine the interpolation position when the Light Probe or Reflection Probe systems are used.

Details

The **Render Mode** lets you choose between several 2D Billboard graphic modes and Mesh mode. Using 3D Meshes gives particles extra authenticity when they represent solid GameObjects, such as rocks, and can also improve the sense of volume for clouds, fireballs and liquids. Meshes must be read/write enabled to work on the Particle System. If you assign them in the editor Unity handles this for you but if you want to assign different meshes at runtime you need to handle this setting yourself.

When 2D billboard graphics are used, the different options can be used for a variety of effects:

Billboard mode is good for particles representing volumes that look much the same from any direction (such as clouds).

Horizontal Billboard mode can be used when the particles cover the ground (such as target indicators and magic spell effects) or when they are flat objects that fly or float parallel to the ground (for example, shurikens).

Vertical Billboard mode keeps each particle upright and perpendicular to the XZ plane, but allows it to rotate around its y-axis. This can be helpful when you are using an orthographic Camera and want particle sizes to stay consistent.

Stretched Billboard mode accentuates the apparent speed of particles in a similar way to the “stretch and squash” techniques used by traditional animators. Note that in Stretched Billboard mode, particles are aligned to face the Camera and also aligned to their velocity. This alignment occurs regardless of the Velocity Scale value - even if Velocity Scale is set to 0, particles in this mode still align to the velocity.

The Normal Direction can be used to create spherical shading on the flat rectangular billboards. This can help create the illusion of 3D particles if you are using a Material that applies lighting to your particles. This setting is only used with the Billboard render modes.

2018–03–28 Page published with editorial review

GPU instancing added in Unity 2018.1

Particle Systems (Legacy, prior to release 3.5)

Leave feedback

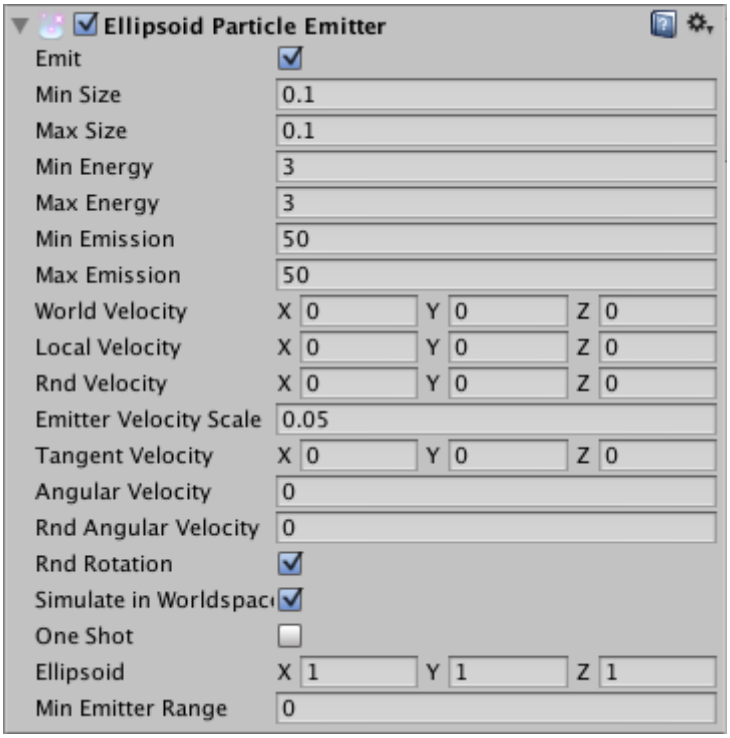
Particles are essentially 2D images rendered in 3D space. They are primarily used for effects such as smoke, fire, water droplets, or leaves. A **Particle System** is made up of three separate **Components**: **Particle Emitter**, **Particle Animator**, and a **Particle Renderer**. You can use a Particle Emitter and Renderer together if you want static particles. The Particle Animator will move particles in different directions and change colors. You also have access to each individual particle in a particle system via scripting, so you can create your own unique behaviors that way if you choose.

This section describes the various components available in the legacy particle system.

Ellipsoid Particle Emitter (Legacy)

Leave feedback

The **Ellipsoid Particle Emitter** spawns particles inside a sphere. Use the **Ellipsoid** property below to scale & stretch the sphere.



Properties

Property:	*Function:*
Emit	If enabled, the emitter will emit particles.
Min Size	The minimum size each particle can be at the time when it is spawned.
Max Size	The maximum size each particle can be at the time when it is spawned.
Min Energy	The minimum lifetime of each particle, measured in seconds.
Max Energy	The maximum lifetime of each particle, measured in seconds.
Min Emission	The minimum number of particles that will be spawned every second.
Max Emission	The maximum number of particles that will be spawned every second.
World Velocity	The starting speed of particles in world space, along X, Y, and Z.
Local Velocity	The starting speed of particles along X, Y, and Z, measured in the object's orientation.
Rnd Velocity	A random speed along X, Y, and Z that is added to the velocity.
Emitter Velocity Scale	The amount of the emitter's speed that the particles inherit.
Tangent Velocity	The starting speed of particles along X, Y, and Z, across the Emitter's surface.
Angular Velocity	The angular velocity of new particles in degrees per second.
Rnd Angular Velocity	A random angular velocity modifier for new particles.
Rnd Rotation	If enabled, the particles will be spawned with random rotations.

Property:	*Function:*
Simulate In World Space	If enabled, the particles don't move when the emitter moves. If false, when you move the emitter, the particles follow it around.
One Shot	If enabled, the particle numbers specified by min & max emission is spawned all at once. If disabled, the particles are generated in a long stream.
Ellipsoid	Scale of the sphere along X, Y, and Z that the particles are spawned inside.
MinEmitterRange	Determines an empty area in the center of the sphere - use this to make particles appear on the edge of the sphere.

Details

Ellipsoid Particle Emitters (EPEs) are the basic emitter, and are included when you choose to add a **Particle System** to your scene from **Components->Particles->Particle System**. You can define the boundaries for the particles to be spawned, and give the particles an initial velocity. From here, use the Particle Animator to manipulate how your particles will change over time to achieve interesting effects.

Particle Emitters work in conjunction with Particle Animators and Particle Renderers to create, manipulate, and display Particle Systems. All three **Components** must be present on an object before the particles will behave correctly. When particles are being emitted, all different velocities are added together to create the final velocity.

Spawning Properties

Spawning properties like **Size**, **Energy**, **Emission**, and **Velocity** will give your particle system distinct personality when trying to achieve different effects. Having a small **Size** could simulate fireflies or stars in the sky. A large **Size** could simulate dust clouds in a musky old building.

Energy and **Emission** will control how long your particles remain onscreen and how many particles can appear at any one time. For example, a rocket might have high **Emission** to simulate density of smoke, and high **Energy** to simulate the slow dispersion of smoke into the air.

Velocity will control how your particles move. You might want to change your **Velocity** in scripting to achieve interesting effects, or if you want to simulate a constant effect like wind, set your X and Z **Velocity** to make your particles blow away.

Simulate in World Space

If this is disabled, the position of each individual particle will always translate relative to the **Position** of the emitter. When the emitter moves, the particles will move along with it. If you have **Simulate in World Space** enabled, particles will not be affected by the translation of the emitter. For example, if you have a fireball that is spurting flames that rise, the flames will be spawned and float up in space as the fireball gets further away. If **Simulate in World Space** is disabled, those same flames will move across the screen along with the fireball.

Emitter Velocity Scale

This property will only apply if **Simulate in World Space** is enabled.

If this property is set to 1, the particles will inherit the exact translation of the emitter at the time they are spawned. If it is set to 2, the particles will inherit double the emitter's translation when they are spawned. 3 is triple the translation, **etc**.

One Shot

One Shot emitters will create all particles within the **Emission** property all at once, and cease to emit particles over time. Here are some examples of different particle system uses with **One Shot Enabled** or **Disabled**:

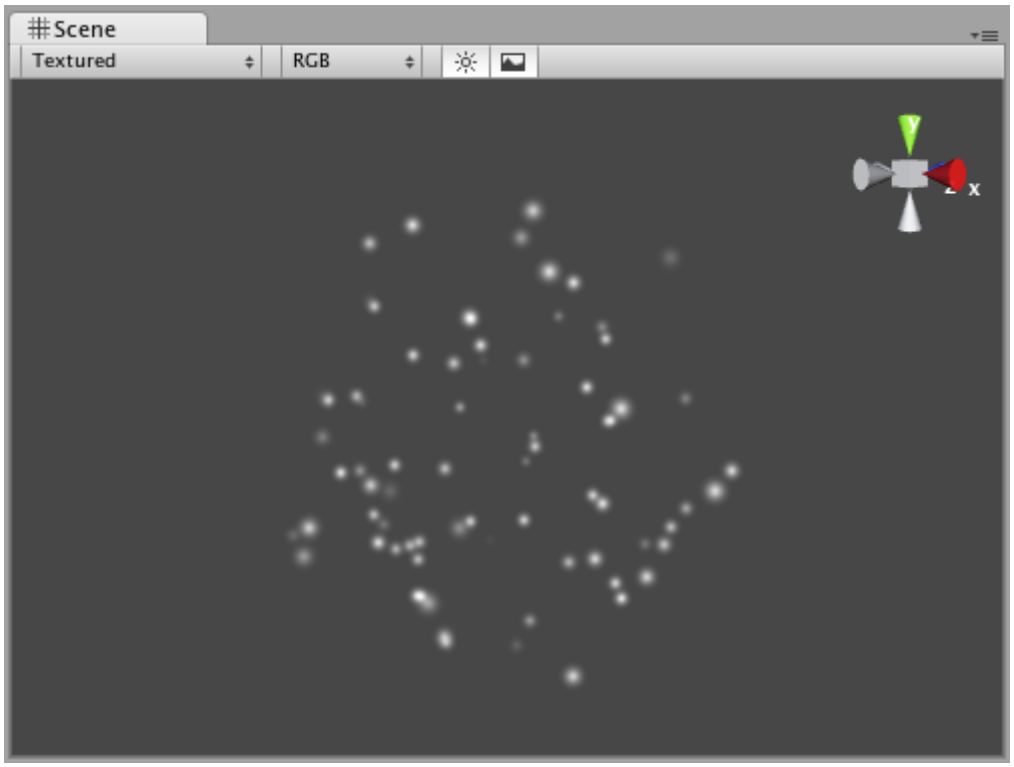
Enabled:

Explosion
Water splash
Magic spell
Disabled:

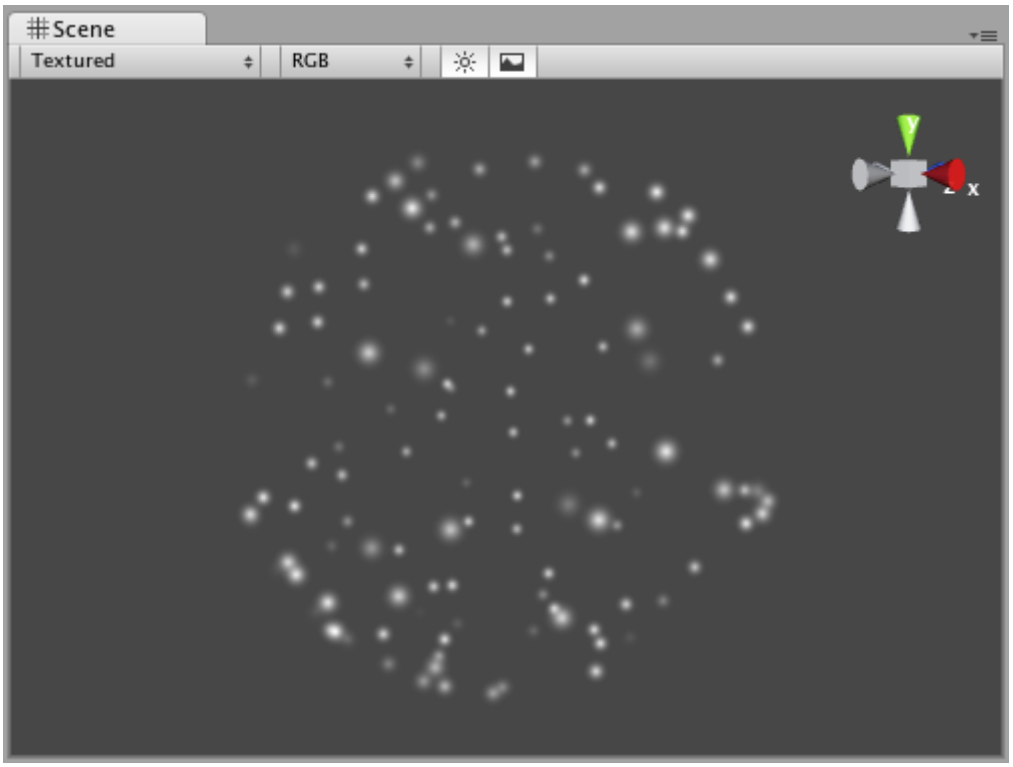
Gun barrel smoke
Wind effect
Waterfall

Min Emitter Range

The **Min Emitter Range** determines the depth within the ellipsoid that particles can be spawned. Setting it to 0 will allow particles to spawn anywhere from the center core of the ellipsoid to the outer-most range. Setting it to 1 will restrict spawn locations to the outer-most range of the ellipsoid.



Min Emitter Range of 0



Min Emitter Range of 1

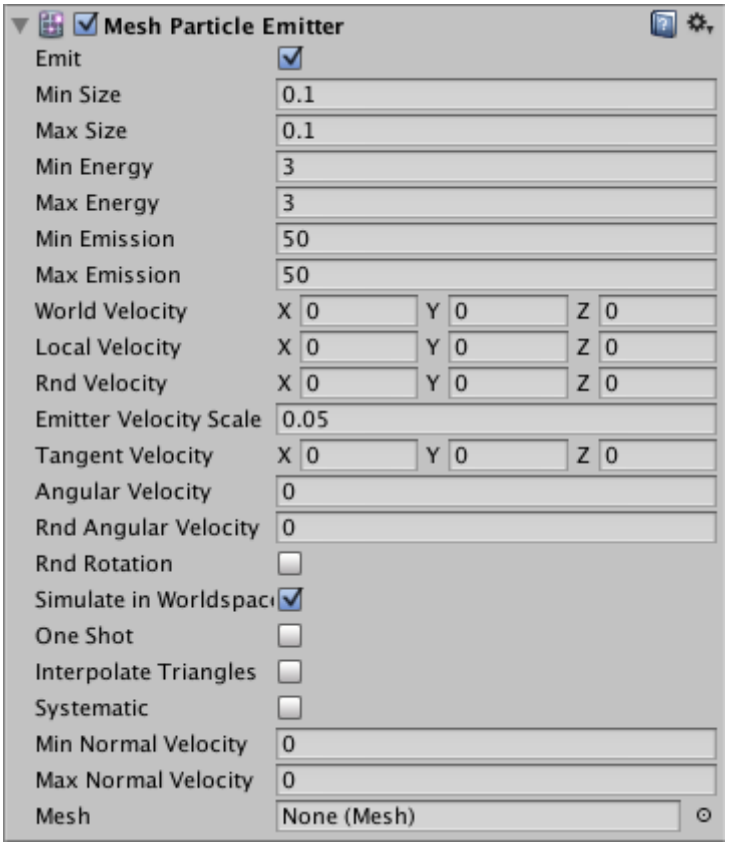
Hints

Be careful of using many large particles. This can seriously hinder performance on low-level machines. Always try to use the minimum number of particles to attain an effect.
The **Emit** property works in conjunction with the **AutoDestruct** property of the Particle Animator. Through scripting, you can cease the emitter from emitting, and then **AutoDestruct** will automatically destroy the Particle System and the **GameObject** it is attached to.

Mesh Particle Emitter (Legacy)

Leave feedback

The **Mesh Particle Emitter** emits particles around a **mesh**. Particles are spawned from the surface of the **mesh**, which can be necessary when you want to make your particles interact in a complex way with objects.



Properties

Property:	*Function:*
Emit	If enabled, the emitter will emit particles.
Min Size	The minimum size each particle can be at the time when it is spawned.
Max Size	The maximum size each particle can be at the time when it is spawned.
Min Energy	The minimum lifetime of each particle, measured in seconds.
Max Energy	The maximum lifetime of each particle, measured in seconds.
Min Emission	The minimum number of particles that will be spawned every second.
Max Emission	The maximum number of particles that will be spawned every second.
World Velocity	The starting speed of particles in world space, along X, Y, and Z.
Local Velocity	The starting speed of particles along X, Y, and Z, measured in the object's orientation.
Rnd Velocity	A random speed along X, Y, and Z that is added to the velocity.

Property:	*Function:*
Emitter Velocity Scale	The amount of the emitter's speed that the particles inherit.
Tangent Velocity	The starting speed of particles along X, Y, and Z, across the Emitter's surface.
Angular Velocity	The angular velocity of new particles in degrees per second.
Rnd Angular Velocity	A random angular velocity modifier for new particles.
Rnd Rotation	If enabled, the particles will be spawned with random rotations.
Simulate In World Space	If enabled, the particles don't move when the emitter moves. If false, when you move the emitter, the particles follow it around.
One Shot	If enabled, the particle numbers specified by min & max emission is spawned all at once. If disabled, the particles are generated in a long stream.
Interpolate Triangles	If enabled, particles are spawned all over the mesh's surface. If disabled, particles are only spawned from the mesh's vertrices.
Systematic	If enabled, particles are spawned in the order of the vertices defined in the mesh. Although you seldom have direct control over vertex order in meshes, most 3D modelling applications have a very systematic setup when using primitives. It is important that the mesh contains no faces in order for this to work.
Min Normal Velocity	Minimum amount that particles are thrown away from the mesh.
Max Normal Velocity	Maximum amount that particles are thrown away from the mesh.

Details

Mesh Particle Emitters (MPEs) are used when you want more precise control over the spawn position & directions than the simpler **Ellipsoid Particle Emitter** gives you. They can be used for making advanced effects.

MPEs work by emitting particles at the vertices of the attached mesh. Therefore, the areas of your mesh that are more dense with polygons will be more dense with particle emission.

Particle Emitters work in conjunction with Particle Animators and Particle Renderers to create, manipulate, and display **Particle Systems**. All three **Components** must be present on an object before the particles will behave correctly. When particles are being emitted, all different velocities are added together to create the final velocity.

Spawning Properties

Spawning properties like **Size**, **Energy**, **Emission**, and **Velocity** will give your particle system distinct personality when trying to achieve different effects. Having a small **Size** could simulate fireflies or stars in the sky. A large **Size**

could simulate dust clouds in a musky old building.

Energy and **Emission** will control how long your particles remain onscreen and how many particles can appear at any one time. For example, a rocket might have high **Emission** to simulate density of smoke, and high **Energy** to simulate the slow dispersion of smoke into the air.

Velocity will control how your particles move. You might want to change your **Velocity** in scripting to achieve interesting effects, or if you want to simulate a constant effect like wind, set your X and Z **Velocity** to make your particles blow away.

Simulate in World Space

If this is disabled, the position of each individual particle will always translate relative to the **Position** of the emitter. When the emitter moves, the particles will move along with it. If you have **Simulate in World Space** enabled, particles will not be affected by the translation of the emitter. For example, if you have a fireball that is spurting flames that rise, the flames will be spawned and float up in space as the fireball gets further away. If **Simulate in World Space** is disabled, those same flames will move across the screen along with the fireball.

Emitter Velocity Scale

This property will only apply if **Simulate in World Space** is enabled.

If this property is set to 1, the particles will inherit the exact translation of the emitter at the time they are spawned. If it is set to 2, the particles will inherit double the emitter's translation when they are spawned. 3 is triple the translation, **etc**.

One Shot

One Shot emitters will create all particles within the **Emission** property all at once, and cease to emit particles over time. Here are some examples of different particle system uses with **One Shot Enabled** or **Disabled**:

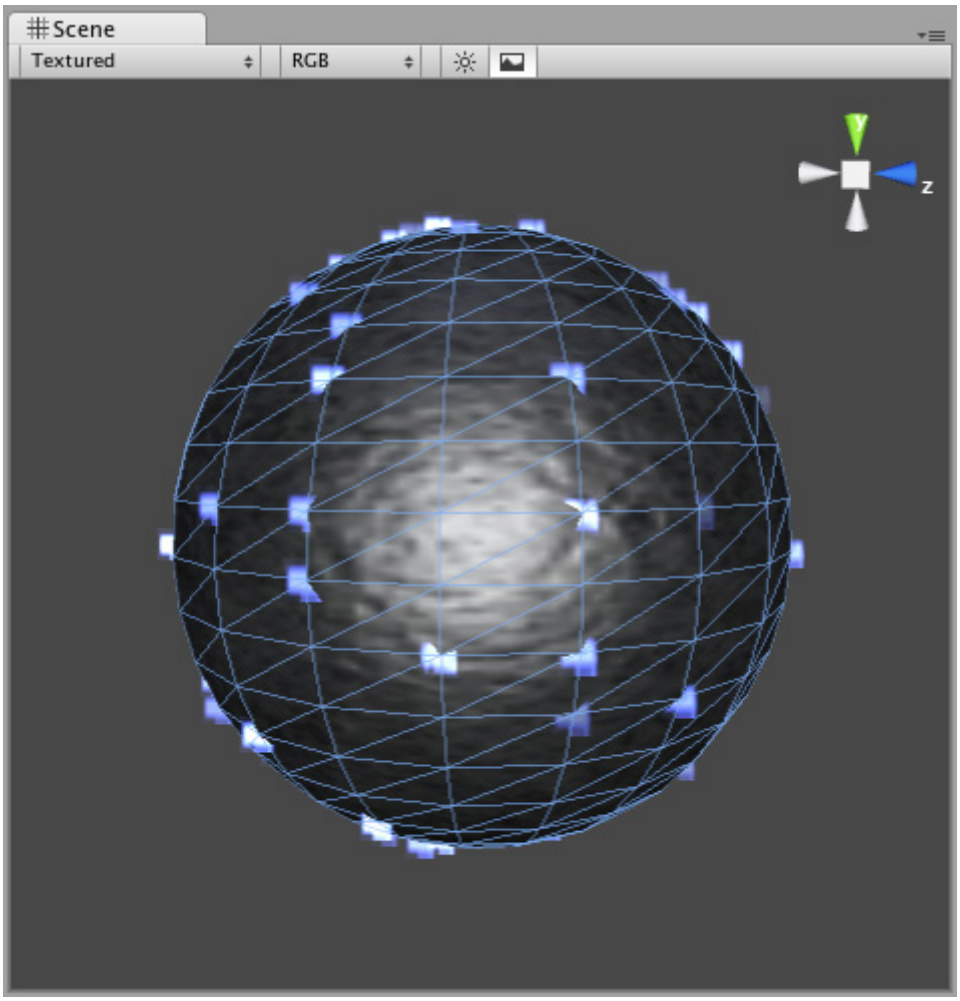
Enabled:

Explosion
Water splash
Magic spell
Disabled:

Gun barrel smoke
Wind effect
Waterfall

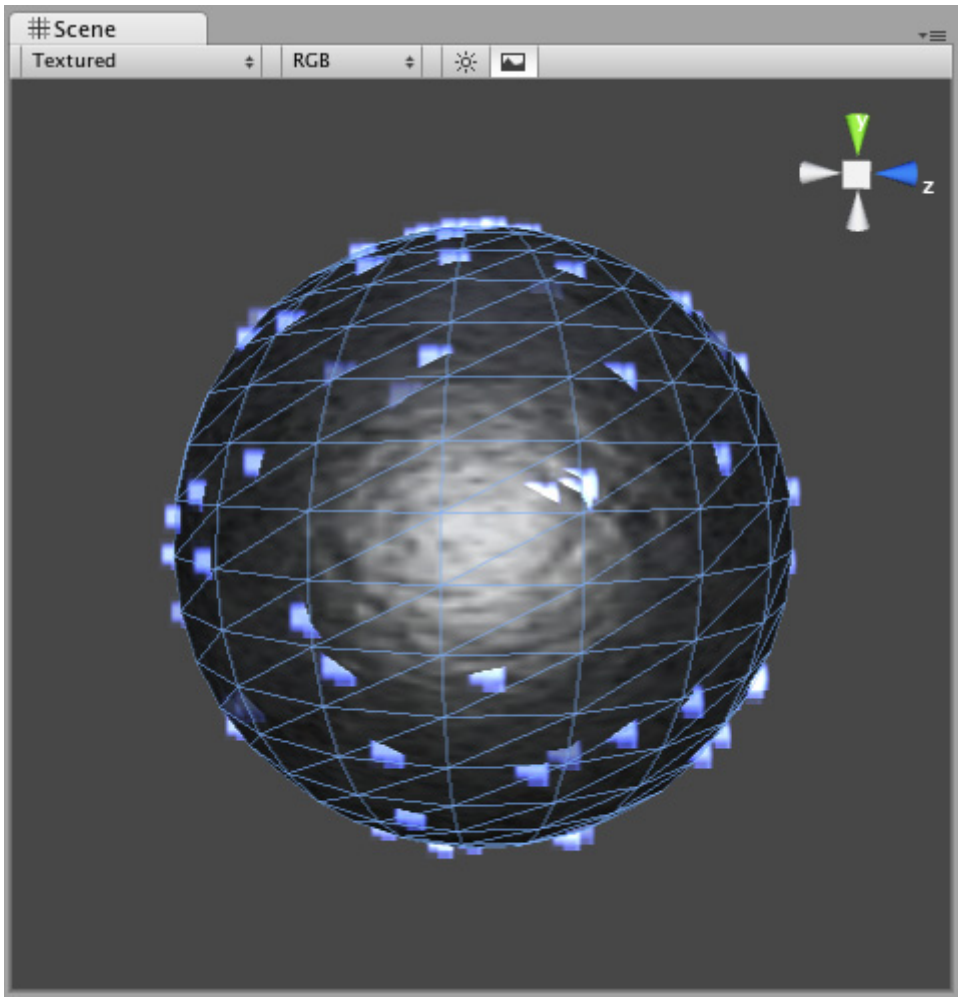
Interpolate Triangles

Enabling your emitter to **Interpolate Triangles** will allow particles to be spawned between the mesh's vertices. This option is off by default, so particles will only be spawned at the vertices.



A sphere with Interpolate Triangles off (the default)

Enabling this option will spawn particles on and in-between vertices, essentially all over the mesh's surface (seen below).

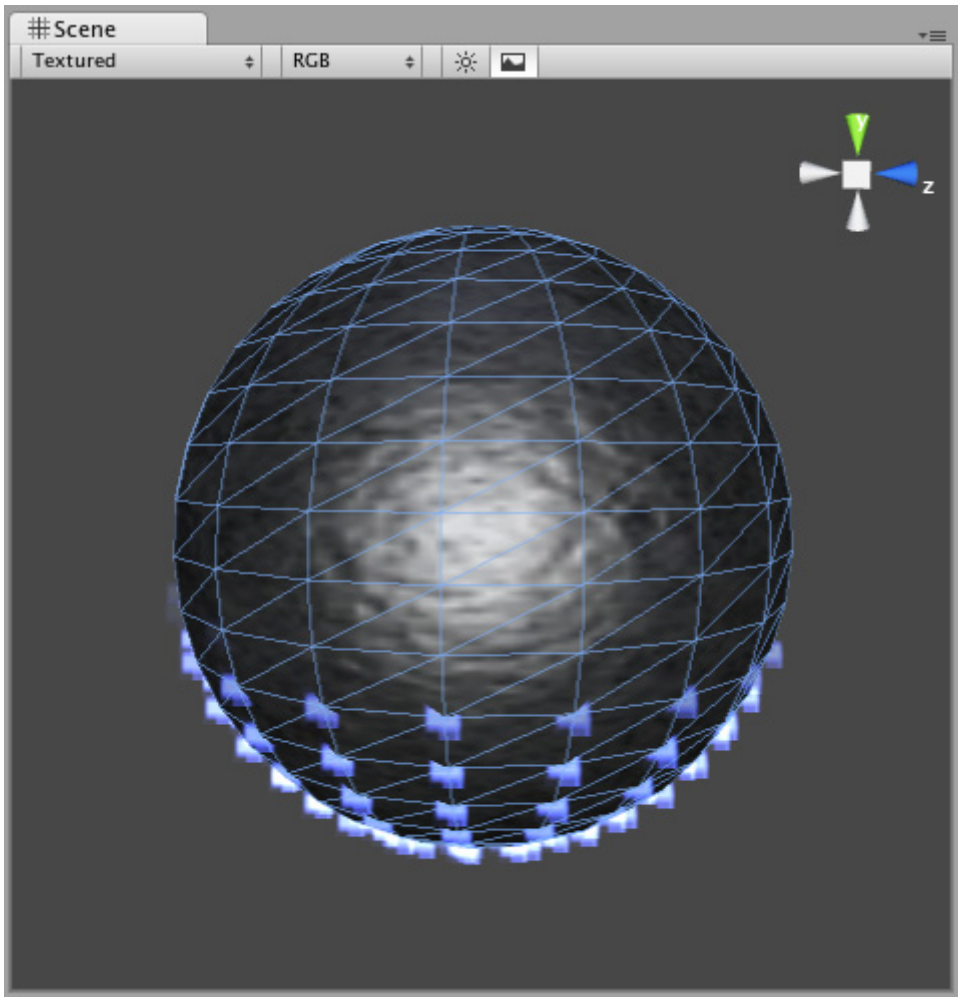


A sphere with Interpolate Triangles on

It bears repeating that even with **Interpolate Triangles** enabled, particles will still be denser in areas of your mesh that are more dense with polygons.

Systematic

Enabling **Systematic** will cause your particles to be spawned in your mesh's vertex order. The vertex order is set by your 3D modeling application.



An MPE attached to a sphere with Systematic enabled

Normal Velocity

Normal Velocity controls the speed at which particles are emitted along the normal from where they are spawned.

For example, create a Mesh Particle System, use a cube mesh as the emitter, enable **Interpolate Triangles**, and set **Normal Velocity Min** and **Max** to 1. You will now see the particles emit from the faces of the cube in a straight line.

See Also

How to make a Mesh Particle Emitter

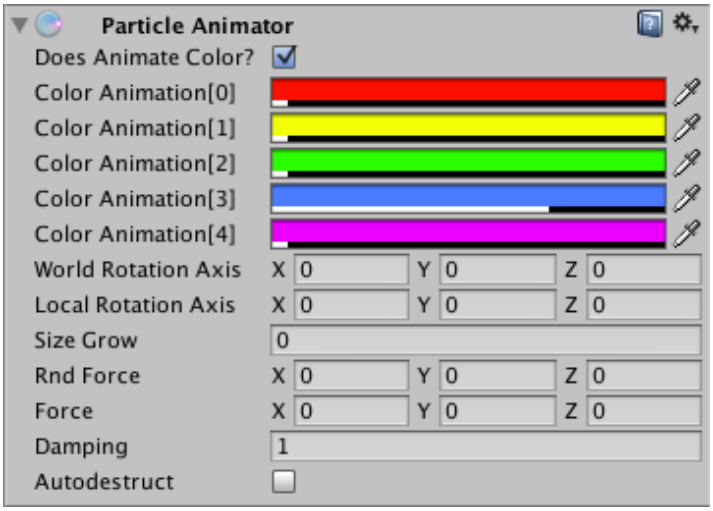
Hints

Be careful of using many large particles. This can seriously hinder performance on low-level machines. Always try to use the minimum number of particles to attain an effect.
The **Emit** property works in conjunction with the **AutoDestruct** property of the Particle Animator. Through scripting, you can cease the emitter from emitting, and then **AutoDestruct** will automatically destroy the Particle System and the **GameObject** it is attached to.
MPEs can also be used to make glow from a lot of lamps placed in a **scene**. Simply make a mesh with one vertex in the center of each lamp, and build an MPE from that with a halo material. Great for evil sci-fi worlds.

Particle Animator (Legacy)

Leave feedback

Particle Animators move your particles over time, you use them to apply wind, drag & color cycling to your **particle systems**.



Properties

Property:	*Function:*
Does Animate Color	If enabled, particles cycle their color over their lifetime.
Color Animation	The 5 colors particles go through. All particles cycle over this - if some have a shorter life span than others, they will animate faster.
World Rotation Axis	An optional world-space axis the particles rotate around. Use this to make advanced spell effects or give caustic bubbles some life.
Local Rotation Axis	An optional local-space axis the particles rotate around. Use this to make advanced spell effects or give caustic bubbles some life.
Size Grow	Use this to make particles grow in size over their lifetime. As randomized forces will spread your particles out, it is often nice to make them grow in size so they don't fall apart. Use this to make smoke rise upwards, to simulate wind, **etc**.
Rnd Force	A random force added to particles every frame. Use this to make smoke become more alive.
Force	The force being applied every frame to the particles, measure relative to the world.
Damping	How much particles are slowed every frame. A value of 1 gives no damping, while less makes them slow down.
Autodestruct	If enabled, the GameObject attached to the Particle Animator will be destroyed when all particles disappear.

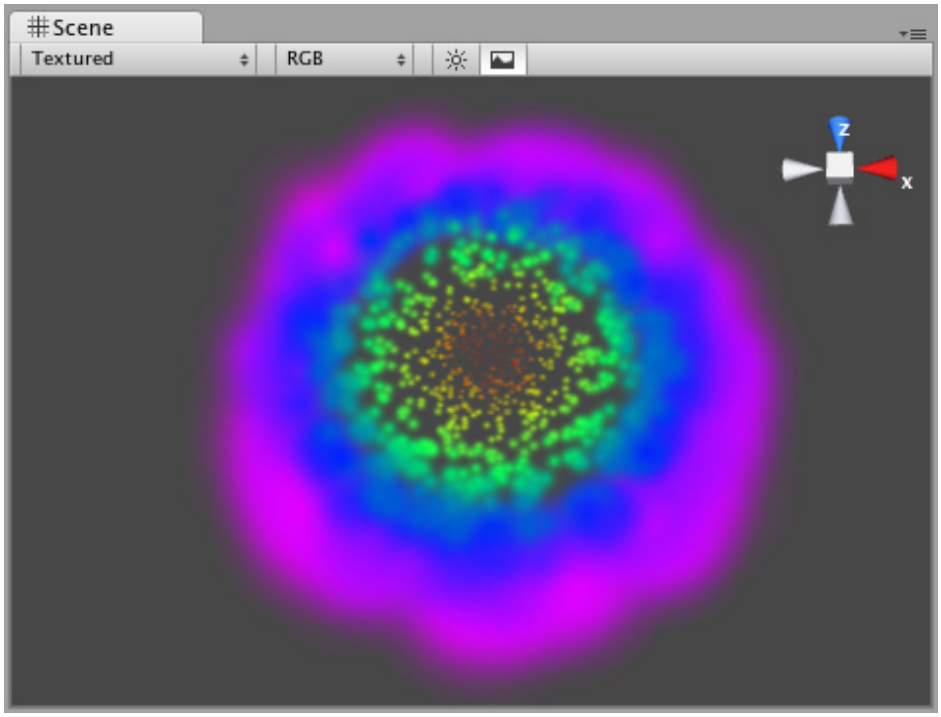
Details

Particle Animators allow your particle systems to be dynamic. They allow you to change the color of your particles, apply forces and rotation, and choose to destroy them when they are finished emitting. For more information about Particle Systems, reference Mesh Particle Emitters, Ellipsoid Particle Emitters, and Particle Renderers.

Animating Color

If you would like your particles to change colors or fade in/out, enable them to **Animate Color** and specify the colors for the cycle. Any particle system that animates color will cycle through the 5 colors you choose. The speed at which they cycle will be determined by the Emitter's **Energy** value.

If you want your particles to fade in rather than instantly appear, set your first or last color to have a low Alpha value.



An Animating Color Particle System

Rotation Axes

Setting values in either the Local or World **Rotation Axes** will cause all spawned particles to rotate around the indicated axis (with the **Transform's** position as the center). The greater the value is entered on one of these axes, the faster the rotation will be.

Setting values in the Local Axes will cause the rotating particles to adjust their rotation as the Transform's rotation changes, to match its local axes.

Setting values in the World Axes will cause the particles' rotation to be consistent, regardless of the Transform's rotation.

Forces & Damping

You use force to make particles accelerate in the direction specified by the force.

Damping can be used to decelerate or accelerate without changing their direction:

A value of 1 means no **Damping** is applied, the particles will not slow down or accelerate.
A value of 0 means particles will stop immediately.
A value of 2 means particles will double their speed every second.

Destroying GameObjects attached to Particles

You can destroy the Particle System and any attached **GameObject** by enabling the **AutoDestruct** property. For example, if you have an oil drum, you can attach a Particle System that has **Emit** disabled and **AutoDestruct** enabled. On **collision**, you enable the Particle Emitter. The explosion will occur and after it is over, the Particle System and the oil drum will be destroyed and removed from the **scene**.

Note that automatic destruction takes effect only after some particles have been emitted. The precise rules for when the object is destroyed when **AutoDestruct** is on:

If there have been some particles emitted already, but all of them are dead now, *or*
If the emitter did have **Emit** on at some point, but now **Emit** is off.

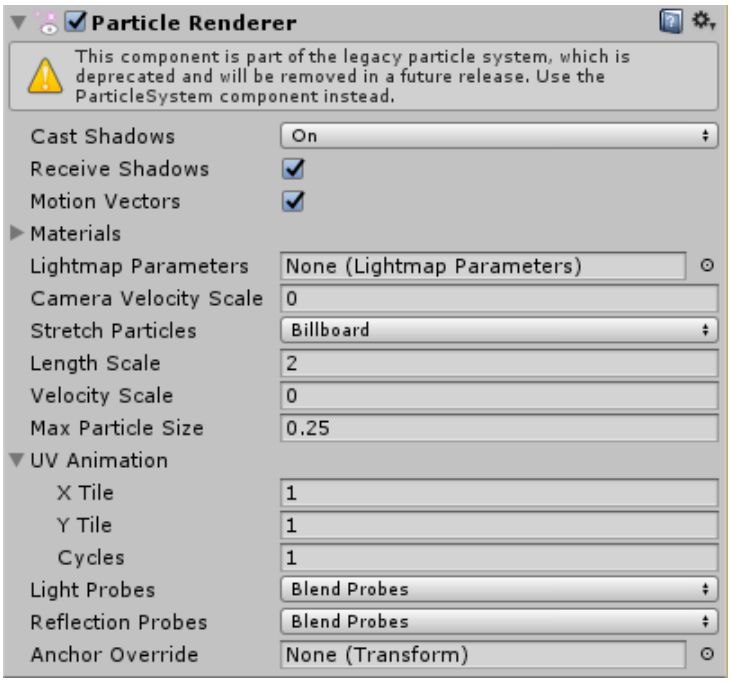
Hints

Use the **Color Animation** to make your particles fade in & out over their lifetime - otherwise, you will get nasty-looking pops.
Use the **Rotation Axes** to make whirlpool-like swirly motions.

Particle Renderer (Legacy)

Leave feedback

The **Particle Renderer** renders the **Particle System** on screen.



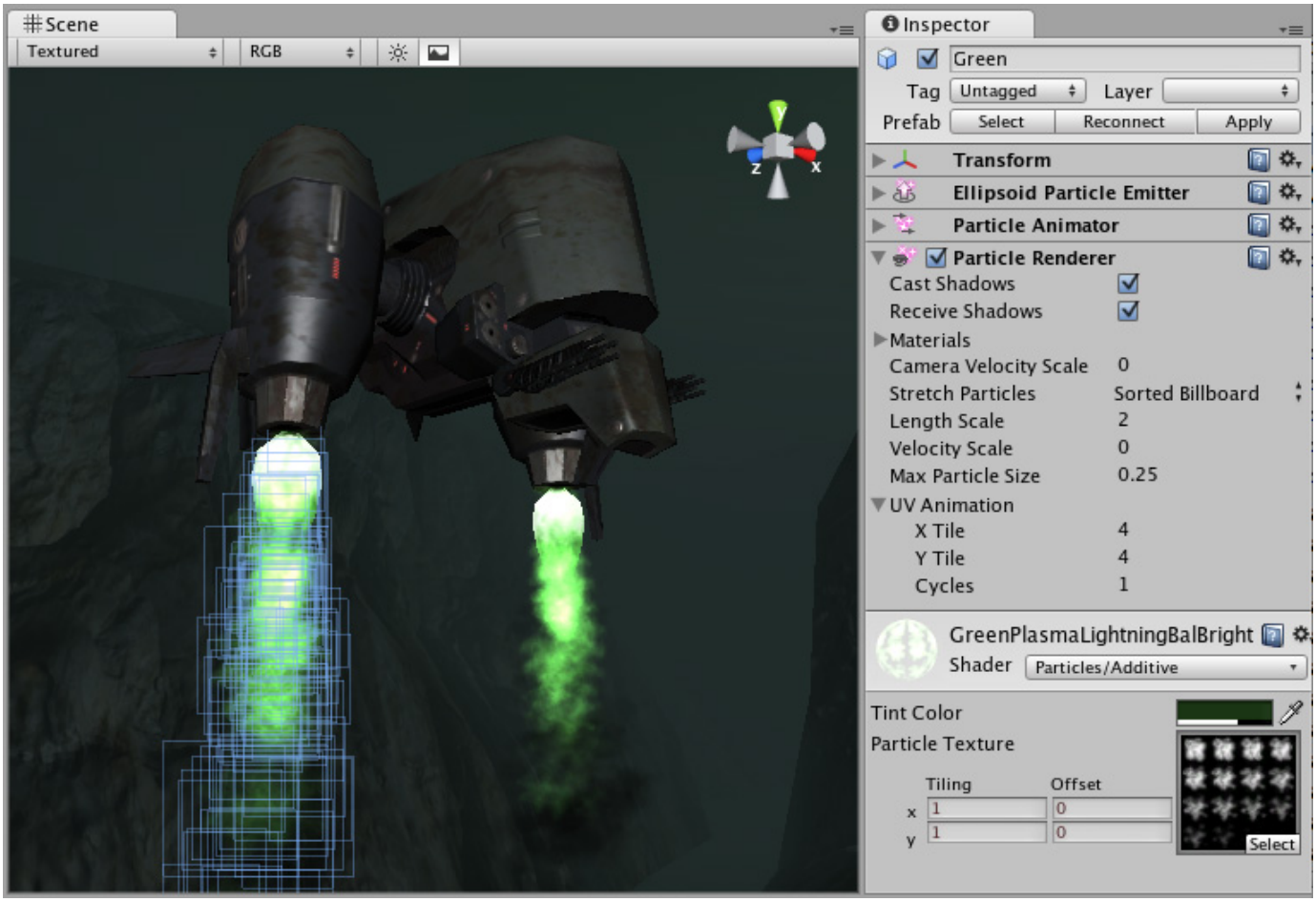
Properties

Property:	*Function:*
Cast Shadows	If enabled, this allows the **Mesh** to cast shadows.
Receive Shadows	If enabled, this allows the Mesh to receive shadows.
Motion Vectors	If enabled, the line has motion vectors rendered into the Camera motion vector Texture. See Renderer.motionVectors in the Scripting API reference documentation to learn more.
Materials	Reference to a list of **Materials** that are displayed in the position of each individual particle.
Light Probes	Probe-based lighting **interpolation** mode.
Reflection Probes	If enabled and reflection probes are present in the **Scene**, a reflection Texture is picked for this Particle Renderer and set as a built-in Shader uniform variable.
Probe Anchor	If defined (using a GameObject), the Renderer uses this **GameObject**'s position to find the interpolated Light Probe.
Camera Velocity Scale	The amount of stretching that is applied to the Particles based on Camera movement.
Stretch Particles	Determines how the particles are rendered:
Billboard	The particles are rendered as if facing the Camera.
Stretched	The particles are facing the direction they are moving.
SortedBillboard	The particles are sorted by depth. Use this when using a blending Material.
VerticalBillboard	All particles are aligned flat along the X/Z axes.
HorizontalBillboard	All particles are aligned flat along the X/Y axes.
Length Scale	If **Stretch Particles** is set to **Stretched**, this value determines how long the particles are in their direction of motion.
Velocity Scale	If **Stretch Particles** is set to **Stretched**, this value determines the rate at which particles are stretched, based on their movement speed.

Property:	*Function:*
UV Animation	If X, Y or both are defined, the UV coordinates of the particles are generated for use with a tile animated texture. See Animated Textures, below.
X Tile	Number of frames located across the X axis.
Y Tile	Number of frames located across the Y axis.
Cycles	How many times to loop the animation sequence.

Details

Particle Renderers are required for any Particle System to be displayed on the screen.



A Particle Renderer makes the Gunship's engine exhaust appear on the screen

Choosing a Material

When setting up a Particle Renderer, it is very important to use an appropriate Material and Shader that renders both sides of the Material. Unity recommends using Particle Shaders with the Particle Renderer.; most of the time you can simply use a Material with one of the built-in Particle Shaders. There are some premade Materials in the **Standard Assets** > **Particles** > **Sources** folder that you can use.

Creating a new Material is easy:

In the Unity menu bar, go to **Assets** > **Create** > **Material**.
In the Inspector window, navigate to **New Material** and click the **Shader** dropdown. Choose one of the Shaders in the **Particles** group (for example: **Particles** > **Multiply**).
To assign a Texture, navigate to the grey box in the **Inspector** window containing the text **None (Texture)** and click the **Select** button to launch a pop-up menu containing the Textures available.
Note that different Shaders use the alpha channel of the Textures slightly differently, but most of the time a value of black in the alpha channel makes it invisible, and a value of white displays it on screen.

Distorting particles

By default, particles are rendered billboarded (that is, as simple square sprites). This is useful for smoke, explosions, and most other particle effects. See Billboard Renderer for more information.

Particles can be made to stretch with the velocity. **Length Scale** and **Velocity Scale** affects how long the stretched particle is. This is useful for effects like sparks, lightning or laser beams.

Sorted Billboard can be used to make all particles sort by depth. Sometimes this is necessary, mostly when using **Alpha Blended** particle shaders. This can be resource-demanding and affect performance; it should only be used if it makes a significant quality difference when **rendering**.

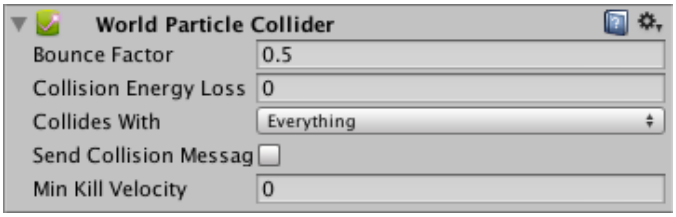
Animated textures

Particle Systems can be rendered with an animated tile Texture. To use this feature, make the Texture out of a grid of images. As the particles go through their life cycle, they cycle through the images. This is good for adding more life to your particles, or making small rotating debris pieces.

World Particle Collider (Legacy)

Leave feedback

The **World Particle Collider** is used to collide particles against other **Colliders** in the **scene**.



Properties

Property:	*Function:*
Bounce Factor	Particles can be accelerated or slowed down when they collide against other objects. This factor is similar to the **Particle Animator's Damping** property.
Collision Energy Loss	Amount of energy (in seconds) a particle should lose when colliding. If the energy goes below 0, the particle is killed.
Min Kill Velocity	If a particle's **Velocity** drops below **Min Kill Velocity** because of a **collision**, it will be eliminated.
Collides with	Which Layers the particle will collide against.
Send Collision Message	If enabled, every particle sends out a collision message that you can catch through scripting.

Details

To create a Particle System with Particle Collider:

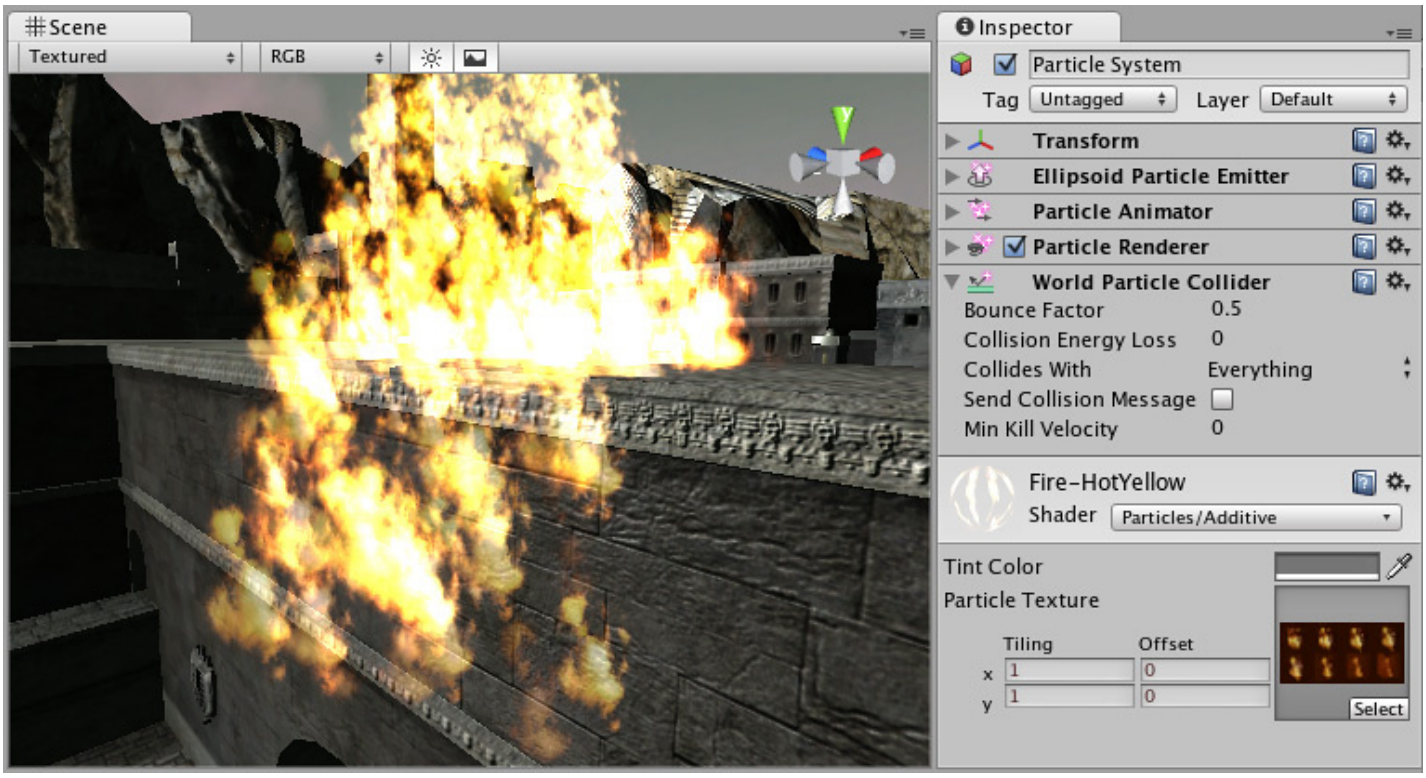
Create a Particle System using **GameObject > Create General > Particle System**
Add the Particle Collider using **Component > Particles > World Particle Collider**

Messaging

If **Send Collision Message** is enabled, any particles that are in a collision will send the message **OnParticleCollision()** to both the particle's **GameObject** and the GameObject the particle collided with.

Hints

Send Collision Message can be used to simulate bullets and apply damage on impact.
Particle **Collision Detection** is slow when used with a lot of particles. Use Particle **Collision Detection** wisely.
Message sending introduces a large overhead and shouldn't be used for normal Particle Systems.



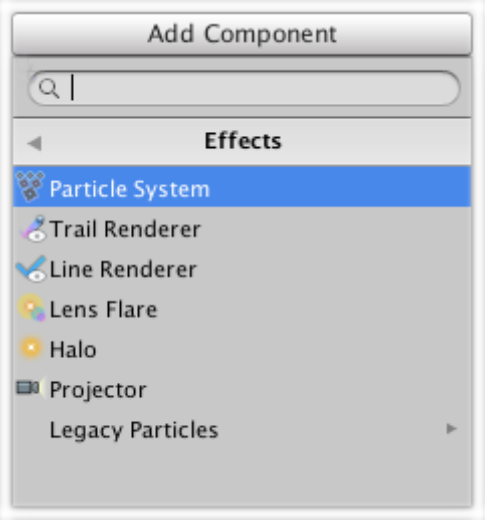
A **Particle System** colliding with a **Mesh Collider**

Visual Effects Reference

Leave feedback

Visual effects can be applied to **cameras**, **GameObjects**, light sources, and other elements of your game. This section provides information on the visual effects available in the Unity Editor.

To access the visual effects available in the Unity Editor, select the item in the Hierarchy window that you wish to apply the effect to, then in the Inspector window go to **Add Component > Effects**.



The visual effects available in the Unity Editor

See the rest of this section for more details on each **component**:

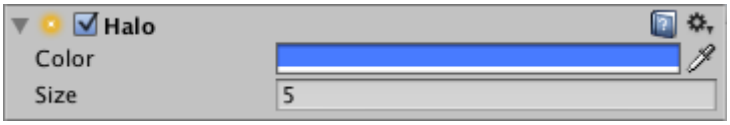
Particle System (see: Particle system in Graphics Reference Documentation)
Trail Renderer
Line Renderer
Lens Flare
Halo
Projector
Legacy Particles (see: Legacy Particles in Graphics Reference Documentation)

Halo

Leave feedback

Halos are light areas around light sources, used to give the impression of small dust particles in the air.

Properties



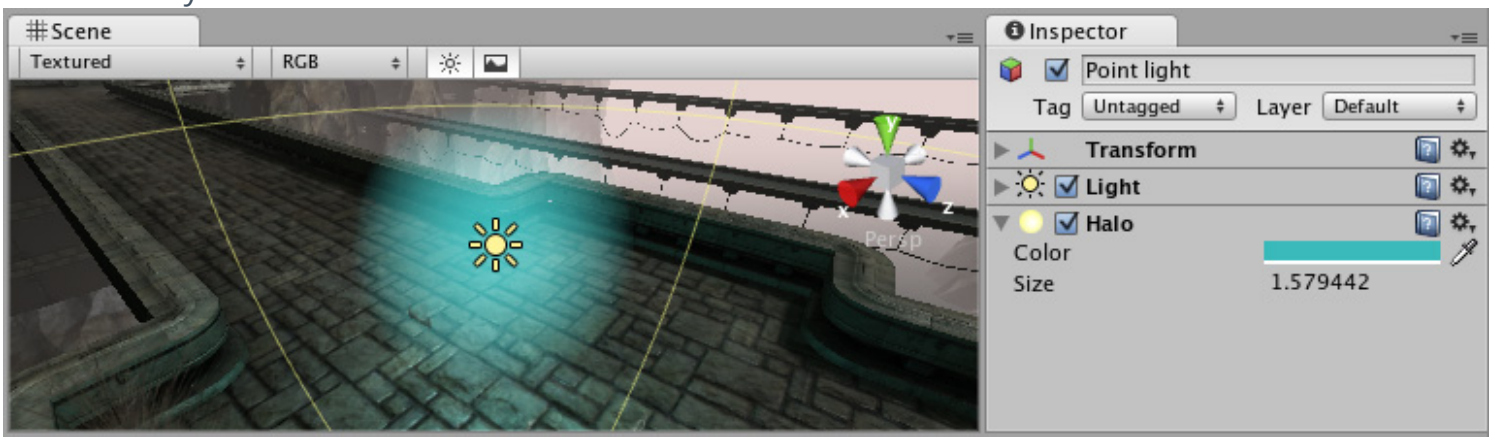
Property:	*Function:*
Color	Color of the Halo.
Size	Size of the Halo.

Details

You can add a Halo component to a Light object and then set its size and color properties to give the desired glowing effect. A Light component can also be set to display a halo without a separate Halo component by enabling its *Draw Halo* property.

Hints

To see Halos in the **scene** view, check **Fx** button in the **Scene View Toolbar**.
To override the **shader** used for Flares, open the Graphics Settings and set **Light Halo** to the shader that you would like to use as the override.



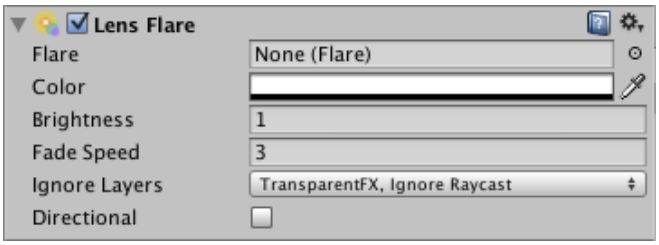
A Light with a separate Halo **Component**

Lens Flare

Leave feedback

SWITCH TO SCRIPTING

Lens Flares simulate the effect of lights refracting inside camera lens. They are used to represent really bright lights or, more subtly, just to add a bit more atmosphere to your **scene**.



The easiest way to setup a Lens Flare is just to assign the Flare property of the Light. Unity contains a couple of pre-configured Flares in the Standard Assets package.

Otherwise, create an empty **GameObject** with **GameObject->Create Empty** from the menu bar and add the Lens Flare **Component** to it with **Component->Effects->Lens Flare**. Then choose the **Flare** in the Inspector.

To see the effect of Lens Flare in the **Scene View**, check the **Effect** drop-down in the Scene View **toolbar** and choose the Flares option.

Enable the Fx button to view Lens Flares in the Scene View

Properties

Property:	*Function:*
Flare	The Flare to render. The flare defines all aspects of the lens flare's appearance.
Color	Some flares can be colorized to better fit in with your scene's mood.
Brightness	How large and bright the Lens Flare is.
Fade Speed	How quickly or slowly the flare will fade.
Ignore Layers	Select masks for layers that shouldn't hide the flare.
Directional	If set, the flare will be oriented along positive Z axis of the game object. It will appear as if it was infinitely far away, and won't track object's position, only the direction of Z axis.

Details

You can directly set flares as a property of a Light Component, or set them up separately as Lens Flare component. If you attach them to a light, they will automatically track the position and direction of the light. To get more precise control, use this Component.

A Camera has to have a Flare Layer Component attached to make Flares visible (this is true by default, so you don't have to do any set-up).

Hints

Be discrete about your usage of Lens Flares.
If you use a very bright Lens Flare, make sure its direction fits with your scene's primary light source.
To design your own Flares, you need to create some Flare **Assets**. Start by duplicating some of the ones we provided in the the **Lens Flares** folder of the **Standard Assets**, then modify from that.

Lens Flares are blocked by **Colliders**. A Collider in-between the Flare GameObject and the Camera will hide the Flare, even if the Collider does not have a **Mesh Renderer**. If the in-between Collider is marked as Trigger it will block the flare if and only if **Physics.queriesHitTriggers** is true.

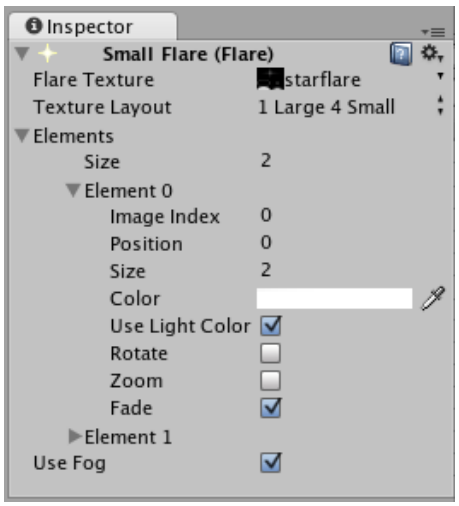
Flare

Leave feedback

SWITCH TO SCRIPTING

Flare objects are the source **assets** that are used by Lens Flare Components. The Flare itself is a combination of a texture file and specific information that determines how the Flare behaves. Then when you want to use the Flare in a **Scene**, you reference the specific Flare from inside a **LensFlare Component** attached to a **GameObject**.

There are some sample Flares in the Standard Assets package. If you want to add one of these to your scene, attach a Lens Flare Component to a GameObject, and drag the Flare you want to use into the **Flare** property of the Lens Flare, just like assigning a **Material** to a **Mesh Renderer**.



The Flare Inspector

Flares work by containing several Flare **Elements** on a single **Texture**. Within the Flare, you pick and choose which **Elements** you want to include from any of the Textures.

Properties

Property:	*Function:*
Elements	The number of Flare images included in the Flare.
Image Index	Which Flare image to use from the **Flare Texture** for this Element. See the *Flare Textures* section below for more information.
Position	The Element's offset along a line running from the containing GameObject's position through the screen center. 0 = GameObject position, 1 = screen center.
Size	The size of the element.
Color	Color tint of the element.
Use Light Color	If the Flare is attached to a Light, enabling this will tint the Flare with the Light's color.
Rotate	If enabled, bottom of the Element will always face the center of the screen, making the Element spin as the Lens Flare moves around on the screen.
Zoom	If enabled, the Element will scale up when it becomes visible and scale down again when it isn't.
Fade	If enabled, the Element will fade in to full strength when it becomes visible and fade out when it isn't.
Flare Texture	A texture containing images used by this Flare's **Elements**. It must be arranged according to one of the **TextureLayout** options.
Texture Layout	How the individual Flare Element images are laid out inside the **Flare Texture** (see *Texture Layouts* below for further details).
Use Fog	If enabled, the Flare will fade away with distance fog. This is used commonly for small Flares.

Details

A Flare consists of multiple **Elements**, arranged along a line. The line is calculated by comparing the position of the GameObject containing the Lens Flare to the center of the screen. The line extends beyond the containing GameObject and the screen center. All Flare **Elements** are strung out on this line.

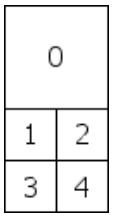
Flare Textures

For performance reasons, all **Elements** of one Flare must share the same Texture. This Texture contains a collection of the different images that are available as Elements in a single Flare. The **Texture Layout** defines how the **Elements** are laid out in the **Flare Texture**.

Texture Layouts

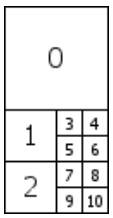
These are the options you have for different Flare **Texture Layouts**. The numbers in the images correspond to the **Image Index** property for each **Element**.

1 Large 4 Small



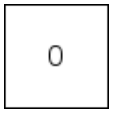
Designed for large sun-style Flares where you need one of the **Elements** to have a higher fidelity than the others. This is designed to be used with Textures that are twice as high as they are wide.

1 Large 2 Medium 8 Small



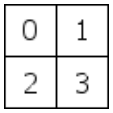
Designed for complex flares that require 1 high-definition, 2 medium and 8 small images. This is used in the standard assets “50mm Zoom Flare” where the two medium Elements are the rainbow-colored circles. This is designed to be used with textures that are twice as high as they are wide.

1 Texture



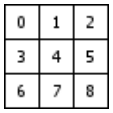
A single image.

2x2 grid



A simple 2x2 grid.

3x3 grid



A simple 3x3 grid.

4x4 grid

0	1	2	3
4	5	...	
	...	14	15

A simple 4x4 grid.

Hints

If you use many different Flares, using a single **Flare Texture** that contains all the **Elements** will give you best **rendering** performance.
Lens Flares are blocked by **Colliders**. A Collider in-between the Flare GameObject and the **Camera** will hide the Flare, even if the Collider does not have a **Mesh Renderer**. If the in-between Collider is marked as Trigger it will block the flare if and only if **Physics.queriesHitTriggers** is true.
To override the **shader** used for Flares, open the Graphics Settings and set **Lens Flares** to the shader that you would like to use as the override.

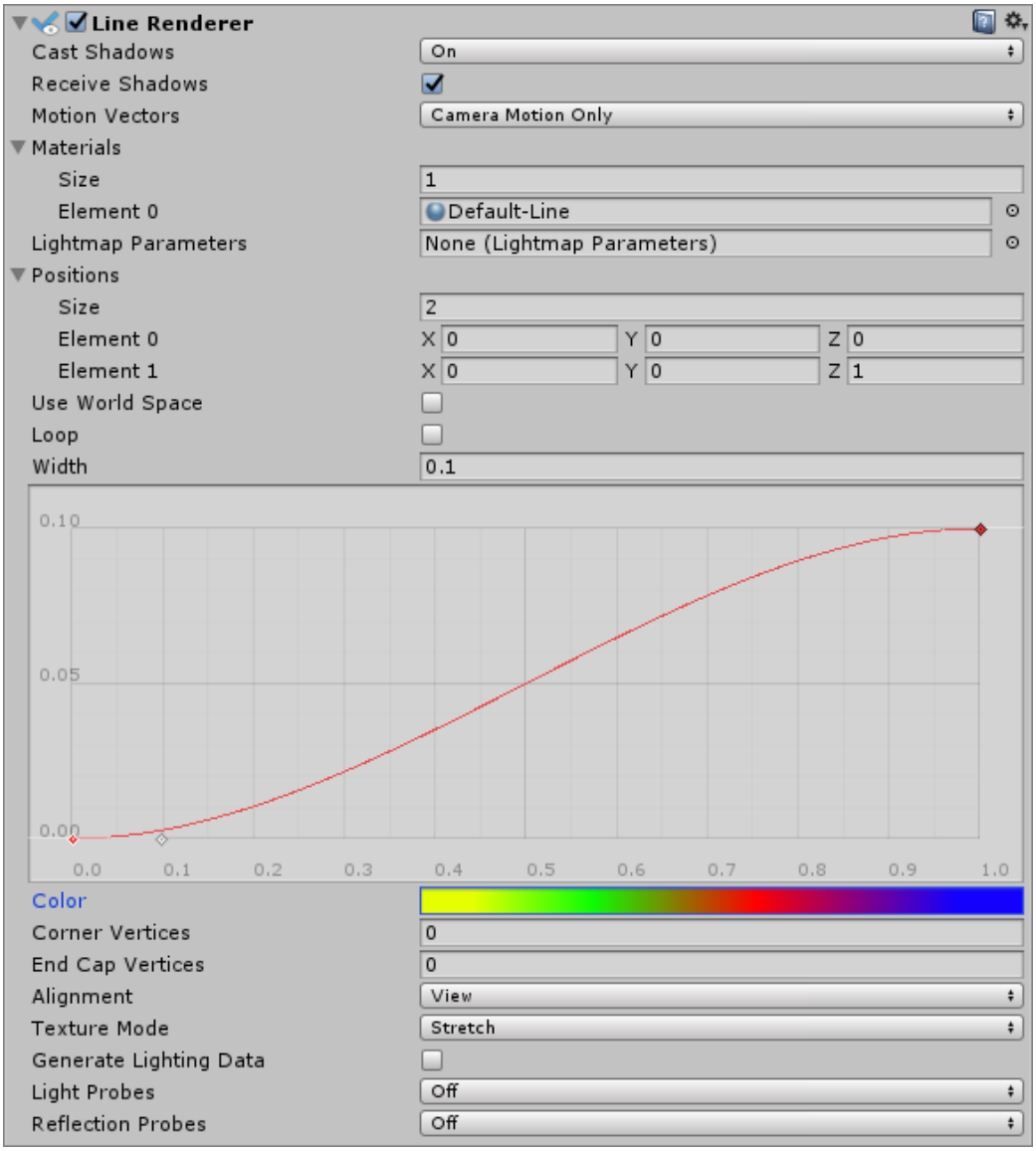
Line Renderer

Leave feedback

SWITCH TO SCRIPTING

The **Line Renderer** component takes an array of two or more points in 3D space, and draws a straight line between each one. A single Line Renderer component can therefore be used to draw anything from a simple straight line to a complex spiral. The line is always continuous; if you need to draw two or more completely separate lines, you should use multiple GameObjects, each with its own Line Renderer.

The Line Renderer does not render one-pixel-wide lines. It renders **billboard** lines (polygons that always face the camera) that have a width in world units and can be textured. It uses the same algorithm for line **rendering** as the Trail Renderer.



Properties

Property	*Function*
Cast Shadows	Determines whether the line casts shadows, whether they should be cast from one or both sides of the line, or whether the line should only cast shadows and not otherwise be drawn. See Renderer.shadowCastingMode in the Scripting API reference documentation to learn more.
Receive Shadows	If enabled, the line receives shadows.

Property	*Function*
Motion Vectors	Select the Motion Vector type to use for this Line Renderer. See Renderer.motionVectorGenerationMode in the Scripting API reference documentation to learn more.
Materials	These properties describe an array of Materials used for rendering the line. The line will be drawn once for each material in the array.
Light Parameters	Reference a Lightmap Parameters **Asset** here to enable the line to interact with the global illumination system.
Positions	These properties describe an array of Vector3 points to connect.
Use World Space	If enabled, the points are considered as world space coordinates, instead of being subject to the transform of the GameObject to which this component is attached.
Loop	Enable this to connect the first and last positions of the line. This forms a closed loop.
Width	Define a width value and a curve to control the width of your line at various points between its start and end. The curve is only sampled at each vertex, so its accuracy is limited by the number of vertices present in your line. The overall width of the line is controlled by the width value.
Color	Define a gradient to control the color of the line along its length.
Corner Vertices	This property dictates how many extra vertices are used when drawing corners in a line. Increase this value to make the line corners appear rounder.
End Cap Vertices	This property dictates how many extra vertices are used to create end caps on the line. Increase this value to make the line caps appear rounder.
Alignment	Set to **View** to make the line face the **Camera**, or **Local** to align it based on the orientation of its **Transform component**.
Texture Mode	Control how the Texture is applied to the line. Use **Stretch** to apply the Texture Map along the entire length of the line, or use **Wrap** to make the Texture repeat along the length of the line. Use the Tiling parameters in the Material to control the repeat rate.
Generate Lighting Data	If enabled (the box is checked), the Line geometry is built with Normals and Tangents included. This allows it to use Materials that use the **scene** lighting, for example via the **Standard Shader**, or by using a custom **shader**.
Light Probes	Probe-based lighting **interpolation** mode.
Reflection Probes	If enabled and reflection probes are present in the Scene, a reflection Texture is picked for this Line Renderer and set as a built-in Shader uniform variable.

Details

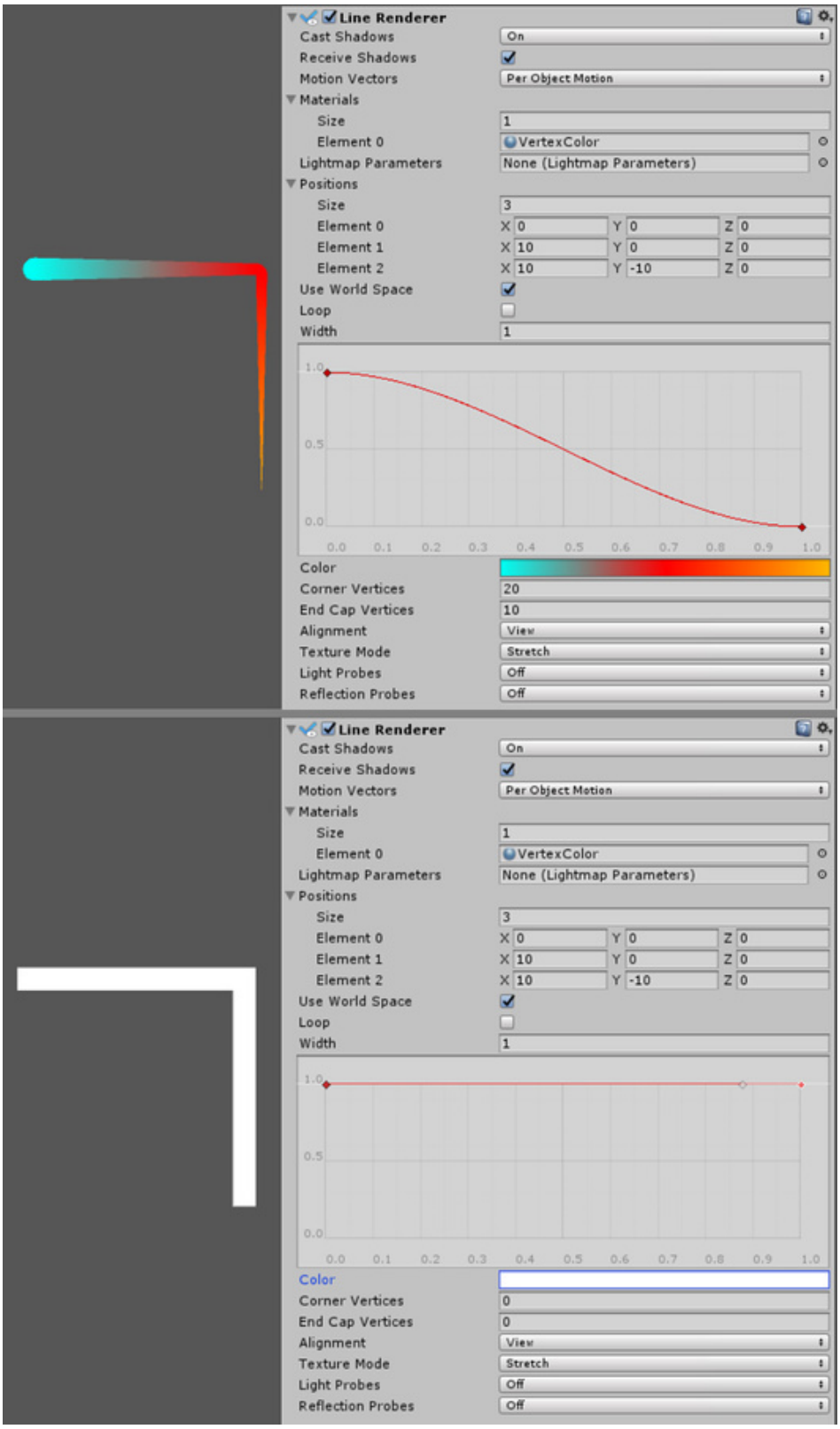
To create a Line Renderer:

In the Unity menu bar, go to **GameObject** > **Create Empty**
In the Unity menu bar, go to **Component** > **Effects** > **Line Renderer**
Drag a Texture or Material onto the Line Renderer. It looks best if you use a Particle Shader in the Material.

Hints

Line Renderers are useful for effects where you need to lay out all the vertices in one frame.
The lines might appear to rotate as you move the Camera. This is intentional when **Alignment** is set to **View**. Set **Alignment** to **Local** to disable this.
The Line Renderer should be the only Renderer on a GameObject.
Unity samples colors from the Color Gradient at each vertex. Between each vertex, Unity applies linear interpolation to colors. Adding more vertices to your Line Renderer might give a closer approximation of a detailed Color Gradient.

Line Renderer example setup



2017–05–31 Page amended with editorial review

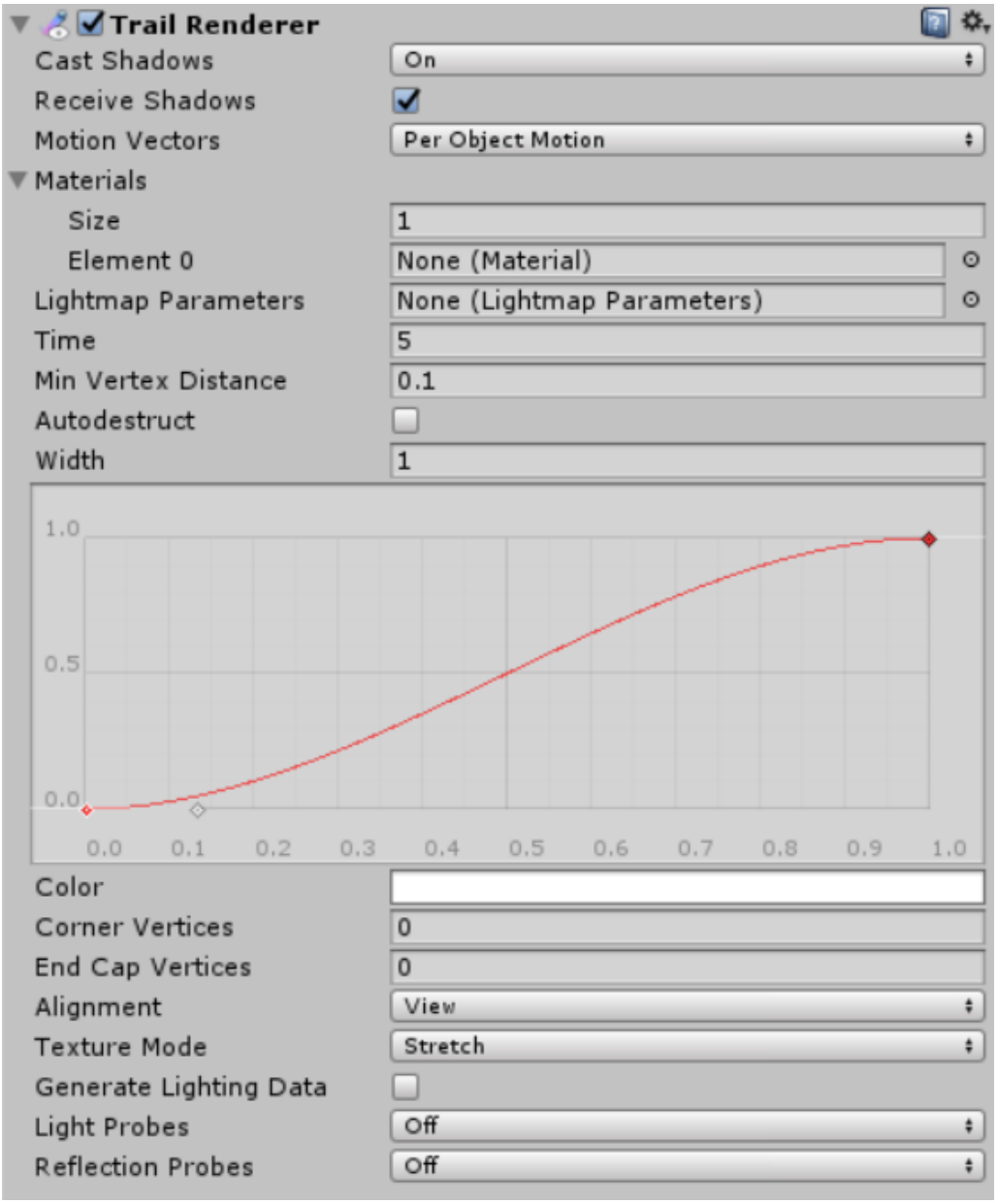
Some properties added in Unity 2017.1

Trail Renderer

Leave feedback

SWITCH TO SCRIPTING

The **Trail Renderer** is used to make trails behind **GameObjects** in the **Scene** as they move.



Properties

Property:	*Function:*
Cast Shadows	Determines whether the trail casts shadows, whether they should be cast from one or both sides of the trail, or whether the trail should only cast shadows and not otherwise be drawn. See Renderer.shadowCastingMode in the Scripting API reference documentation to learn more.
Receive Shadows	If enabled, the trail receives shadows.
Motion Vectors	Select the Motion Vector type to use for this Trail Renderer. See Renderer.motionVectorGenerationMode in the Scripting API reference documentation to learn more.

Property:	*Function:*
Materials	These properties describe an array of Materials used for **rendering** the trail. Particle **Shaders** work best for trails.
Lightmap Parameters	Reference a Lightmap Parameters **Asset** here to enable the trail to interact with the global illumination system.
Time	Define the length of the trail, measured in seconds.
Min Vertex Distance	The minimum distance between anchor points of the trail (see **Minimum vertex separation** below).
AutoDestruct	Enable this to destroy the GameObject once it has been idle for **Time** seconds.
Width	Define a width value and a curve to control the width of your trail at various points between its start and end. The curve is applied from the beginning to the end of the trail, and sampled at each vertex. The overall width of the curve is controlled by the width value.
Color	Define a gradient to control the color of the trail along its length.
Corner Vertices	This property dictates how many extra vertices are used when drawing corners in a trail. Increase this value to make the trail corners appear rounder.
End Cap Vertices	This property dictates how many extra vertices are used to create end caps on the trail. Increase this value to make the trail caps appear rounder.
Alignment	Set to **View** to make the Trail face the **camera**, or **Local** to align it based on the orientation of its Transform **component**.
Texture Mode	Control how the Texture is applied to the Trail. Use **Stretch** to apply the Texture map along the entire length of the trail, or use **Wrap** to repeat the Texture along the length of the Trail. Use the **Tiling** parameters in the Material to control the repeat rate.
Generate Lighting Data	If enabled (the box is checked), the Trail geometry is built with Normals and Tangents included. This allows it to use Materials that use the scene lighting, for example via the **Standard Shader**, or by using a custom shader.
Light Probes	Probe-based lighting **interpolation** mode.
Reflection Probes	If enabled and reflection probes are present in the Scene, a reflection Texture is picked for this Trail Renderer and set as a built-in Shader uniform variable.

Details

The Trail Renderer renders a trail of polygons behind a moving GameObject. This can be used to give an emphasized feeling of motion to a moving object, or to highlight the path or position of moving objects. A trail behind a projectile adds visual clarity to its trajectory; contrails from the tip of a plane's wings are an example of a trail effect that happens in real life.

A Trail Renderer should be the only renderer used on the attached GameObject. It is best to create an empty GameObject, and attach a Trail Renderer as the only renderer. You can then parent the Trail Renderer to whatever GameObject you would like it to follow.

Materials

A Trail Renderer component should use a Material that has a Particle Shader. The Texture used for the Material should be of square dimensions (for example 256x256, or 512x512). The trail is rendered once for each Material present in the array.

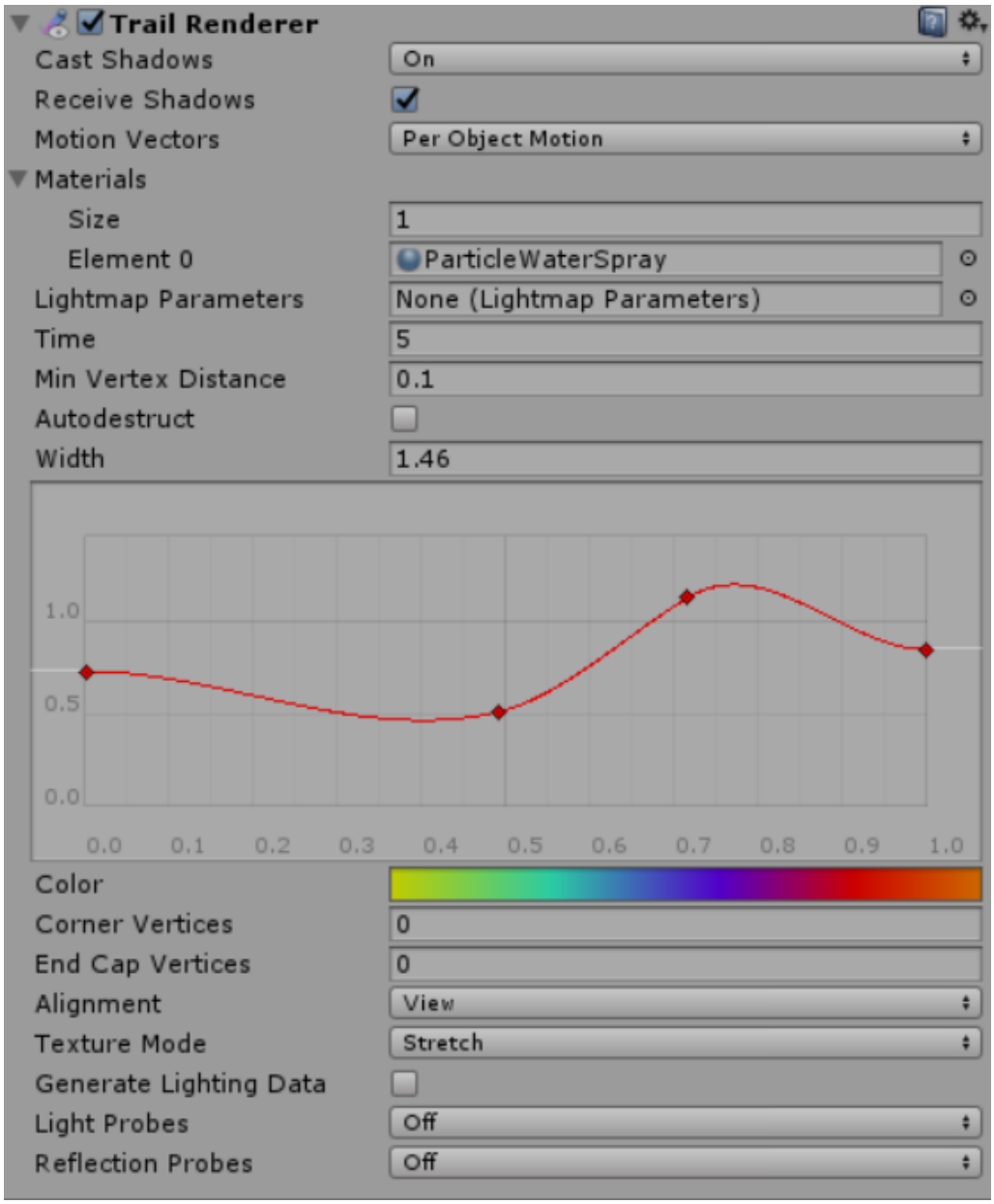
Minimum vertex separation

The **Min Vertex Distance** value determines how far an object that contains a trail must travel before a segment of that trail is solidified. Low values like 0.1 create trail segments more often, creating smoother trails. Higher values like 1.5 create segments that are more jagged in appearance. There is a slight performance trade-off when using lower values/smoother trails, so try to use the largest possible value to achieve the effect you are trying to create. Additionally, wide trails may exhibit visual artifacts when the vertices are very close together and the trail changes direction significantly over a short distance.

Hints

Use Particle Materials with the Trail Renderer.
Trail Renderers must be laid out over a sequence of frames; they cannot appear instantaneously.
Trail Renderers rotate to display the face toward the camera, similar to other **Particle Systems**.
Unity samples colors from the Color Gradient at each vertex. Between each vertex, Unity applies linear interpolation to colors. Adding more vertices to your **Line Renderer** might give a closer approximation of a detailed Color Gradient.

Trail Renderer example setup



A Trail Renderer component as it appears in the Inspector window, set up to create a multicoloured trail that gets thinner and then much wider

The resulting trail created by the above component setup

2017–05–31 Page amended with editorial review

Properties added in Unity 2017.1

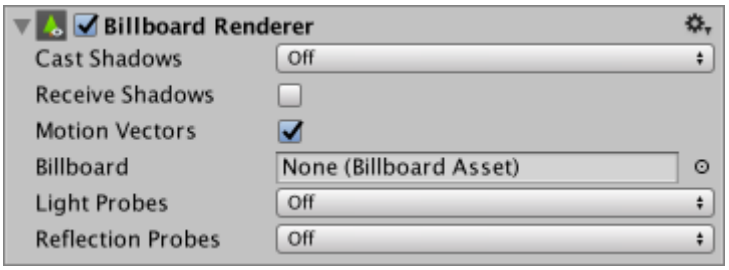
Billboard Renderer

Leave feedback

SWITCH TO SCRIPTING

The **Billboard Renderer** renders BillboardAssets, either from a premade **Asset** (exported from SpeedTree) or from a custom-created file (created using a script at runtime or from a custom editor, for example). For more information about creating **Billboard Assets**, see the BillboardAssets manual page and the BillboardAsset API reference.

Billboards are a level-of-detail (LOD) method of drawing complicated 3D Meshes in a simpler manner when they are distant in a **Scene**. Because the **Mesh** is distant, its size on screen and the low likelihood of it being a focal point in the **Camera** view means there is often less requirement to draw it in full detail.



Property:	Function:
Cast Shadows	If enabled, the billboard creates shadows when a shadow-casting Light shines on it.
On	Enable shadows.
Off	Disable shadows.
Two Sided	Allow shadows to be cast from either side of the billboard (that is, backface culling is not taken into account).
Shadows Only	Show shadows, but not the billboard itself.
Receive Shadows	Check the box to enable shadows to be cast on the billboard.
Motion Vectors	Check the box to enable **rendering** of the billboard's motion vectors into the Camera Motion Vector Texture. See Renderer.motionVectors in the Scripting API for more information.
Billboard	If you have a pre-made Billboard Asset, place it here to assign it to this Billboard Renderer.
Light Probes	If enabled, and if baked Light Probes are present in the Scene, the Billboard Renderer uses an interpolated Light Probe for lighting.
Off	Disable Light Probes.
Blend Probes	The lighting applied to the billboard is interpreted from one interpolated Light Probe.
Use Proxy Volume	The lighting applied to the Billboard Renderer is interpreted from a 3D grid of interpolated Light Probes.

Property:	Function:
Reflection Probes	If enabled, and if Reflection Probes are present in the Scene, a reflection Texture is picked for this **GameObject** and set as a built-in **Shader** uniform variable.
Off	Disable Reflection Probes.
Blend Probes	The reflections applied to the billboard are interpreted from adjacent Reflection Probes, and do not take the the **skybox** into account. This is generally used for GameObjects that are “indoors” or in covered parts of the Scene (such as caves and tunnels), because the sky is not visible and therefore wouldn’t be reflected by the billboard.
Blend Probes and Skybox	This works like Blend Probes, but also allows the skybox to be used in the blending. This is generally used for GameObjects in the open air, where the sky would always be visible and reflected.
Simple	Reflection Probes are enabled, but no blending occurs between probes when there are two overlapping volumes.

Billboard Asset

Leave feedback

SWITCH TO SCRIPTING

A **Billboard Asset** is a collection of pre-rendered images of a more complicated **Mesh** intended for use with the Billboard Renderer, in order to render an object at some distance from a **Camera** at a lower level of detail (LOD) to save on **rendering** time.

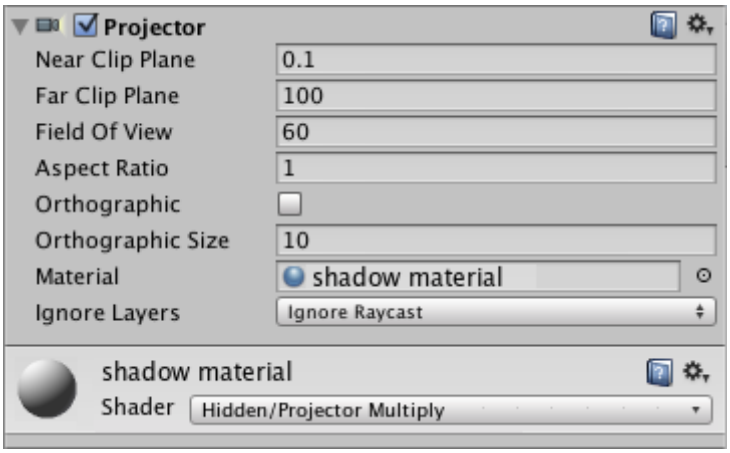
The most common way to generate a **Billboard Asset** is to create files in SpeedTree Modeler, and then import them into Unity. It is also possible to create your own Billboard Assets from script. See the API reference for BillboardAsset for further details.

Projector

Leave feedback

SWITCH TO SCRIPTING

A **Projector** allows you to project a **Material** onto all objects that intersect its frustum. The material must use a special type of **shader** for the projection effect to work correctly - see the projector prefabs in Unity's **standard assets** for examples of how to use the supplied Projector/Light and Projector/Multiply shaders.



Properties

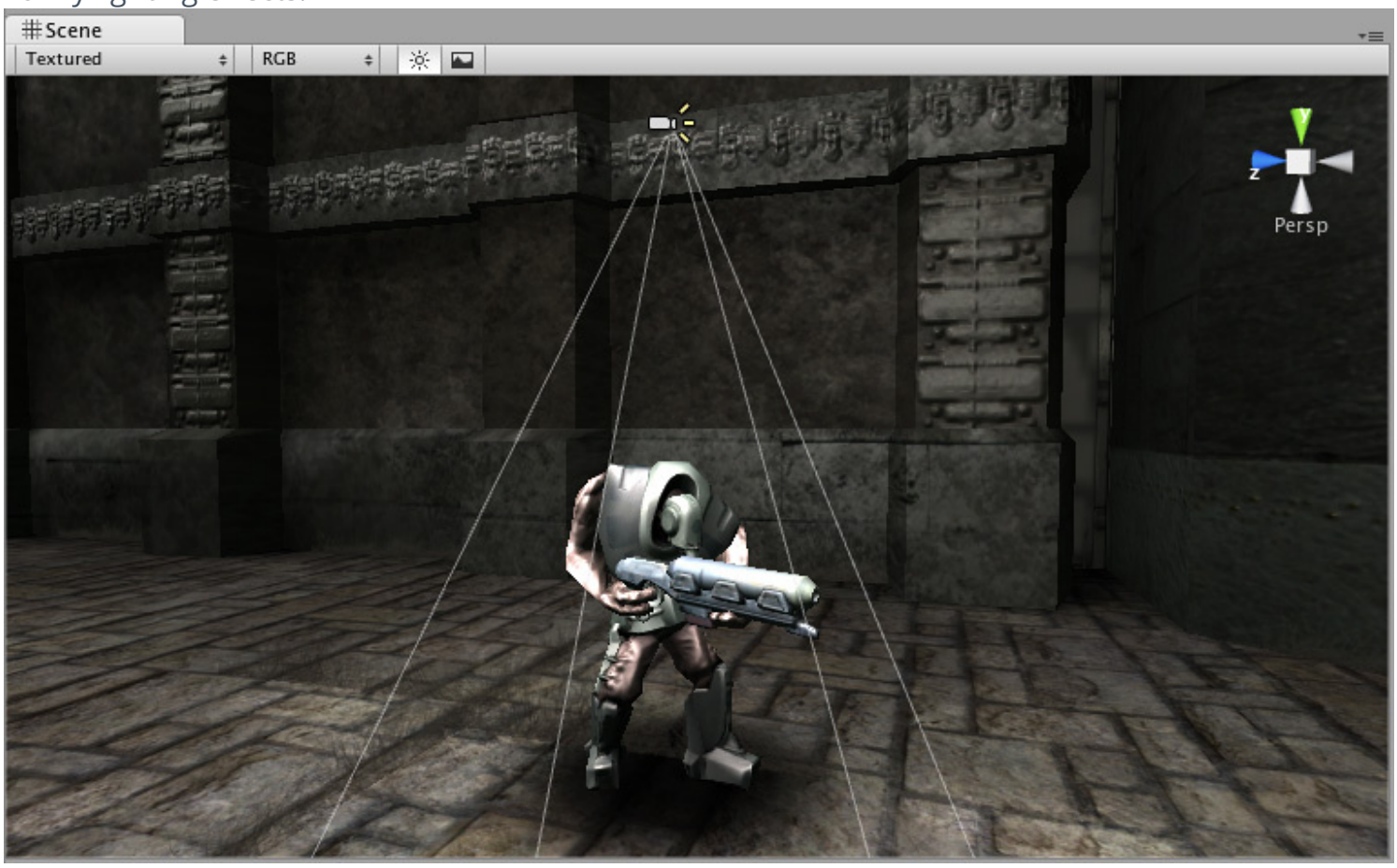
Property:	*Function:*
Near Clip Plane	Objects in front of the near clip plane will not be projected upon.
Far Clip Plane	Objects beyond this distance will not be affected.
Field Of View	The field of view in degrees. This is only used if the Projector is not Ortho Graphic.
Aspect Ratio	The Aspect Ratio of the Projector. This allows you to tune the height vs width of the Projector.
Is Ortho Graphic	If enabled, the Projector will be Ortho Graphic instead of perspective.
Ortho Graphic Size	The Ortho Graphic size of the Projection. this is only used if Is Ortho Graphic is turned on.
Material	The Material that will be Projected onto Objects.
Ignore Layers	Objects that are in one of the Ignore Layers will not be affected. By default, Ignore Layers is none so all geometry that intersects the Projector frustum will be affected.

Details

With a projector you can:

Create shadows.
Make a real world projector on a tripod with another Camera that films some other part of the world using a **Render Texture**.
Create bullet marks.

Funky lighting effects.



A Projector is used to create a Blob Shadow for this Robot

If you want to create a simple shadow effect, simply drag the **StandardAssets->Blob-Shadow->Blob shadow projector Prefab** into your scene. You can modify the Material to use a different Blob shadow texture.

Note: When creating a projector, always be sure to set the wrap mode of the texture's material of the projector to *clamp*. else the projector's texture will be seen repeated and you will not achieve the desired effect of shadow over your character.

Hints

Projector Blob shadows can create very impressive Splinter Cell-like lighting effects if used to shadow the environment properly.

When no **Falloff** Texture is used in the projector's Material, it can project both forward and backward, creating "double projection". To fix this, use an alpha-only Falloff texture that has a black leftmost **pixel** column.

Mesh Components

Leave feedback

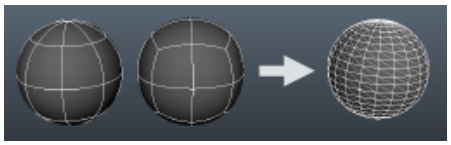
3D **Meshes** are the main graphics primitive of Unity. Various **components** exist in Unity to render regular or skinned meshes, trails or 3D lines.

Meshes

Leave feedback

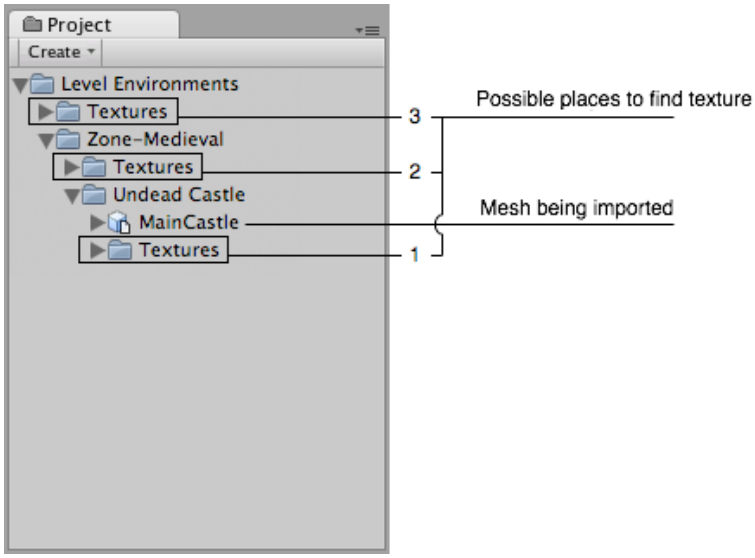
SWITCH TO SCRIPTING

Meshes make up a large part of your 3D worlds. Aside from some Asset store **plugins**, Unity does not include modelling tools. Unity does however have great interactivity with most 3D modelling packages. Unity supports triangulated or Quadrangulated polygon meshes. Nurbs, Nurms, Subdiv surfaces must be converted to polygons.



Textures

Unity will attempt to find the textures used by a **mesh** automatically on import by following a specific search plan. First, the importer will look for a sub-folder called Textures within the same folder as the **mesh** or in any parent folder. If this fails, an exhaustive search of all textures in the project will be carried out. Although slightly slower, the main disadvantage of the exhaustive search is that there could be two or more textures in the project with the same name. In this case, it is not guaranteed that the right one will be found.



Place your textures in a **Textures** folder at or above the asset's level
Material tab of the Import Settings window

Material Generation and Assignment

For each imported material Unity will apply the following rules:-

If material generation is disabled (i.e. **Import Materials** is unchecked), then it will assign the Default-Diffuse material. If it is enabled then it will do the following:

Unity will pick a name for the Unity material based on the **Material Naming** setting
Unity will try to find an existing material with that name. The scope of the Material search is defined by the **Material Search** setting.
If Unity succeeds in finding an existing material then it will use it for the imported **scene**, otherwise it will generate a new material

Colliders

Unity uses two main types of colliders: **Mesh Colliders** and **Primitive Colliders**. Mesh colliders are **components** that use imported mesh data and can be used for environment **collision**. When you enable **Generate Colliders** in the Import Settings, a Mesh

collider is automatically added when the mesh is added to the Scene. It will be considered solid as far as the physics system is concerned.

If you are moving the object around (a car for example), you can not use Mesh colliders. Instead, you will have to use Primitive colliders. In this case you should disable the **Generate Colliders** setting.

Animations

You can import animations from a **Model file**. Follow the guidelines for exporting FBX files from your 3D modeling software before importing it into Unity.

Normal mapping and characters

If you have a character with a normal map that was generated from a high-polygon version of the model, you should import the game-quality version with a **Smoothing angle** of 180 degrees. This will prevent odd-looking seams in lighting due to tangent splitting. If the seams are still present with these settings, enable **Split tangents across UV seams**.

If you are converting a greyscale image into a normal map, you don't need to worry about this.

Blendshapes

Unity has support for BlendShapes (also called morph-targets or vertex level animation). Unity can import BlendShapes from **.FBX** (BlendShapes and controlling aninimation) and **.dae** (only BlendShapes) exported 3D files. Unity BlendShapes support vertex level animation on vertices, normals and tangents. Mesh can be affected by skin and BlendShapes at the same time. All meshes imported with BlendShapes will use SkinnedMeshRenderer (no mater if it does have skin or not). BlendShape animation is imported as part of regular animation - it simply animates BlendShape weights on SkinnedMeshRenderer.

There are two ways to import BlendShapes with normals:

Set **Normals** import mode to **Calculate**, this way same logic will be used for calculating normals on a mesh and BlendShapes.
Export smoothing groups information to the source file. This way, Unity will calculate normals from smoothing groups for mesh and BlendShapes.
If you want tangents on your BlendShapes then set **Tangents** import mode to **Calculate**.

Hints

Merge your meshes together as much as possible. Make them share materials and textures. This has a huge performance benefit.
If you need to set up your objects further in Unity (adding physics, scripts or other coolness), save yourself a world of pain and name your objects properly in your 3D application. Working with lots of *pCube17* or *Box42*-like objects is not fun.
Make your meshes be centered on the world origin in your 3D app. This will make them easier to place in Unity.
If a mesh does not have vertex colors, Unity will automatically add an array of all-white vertex colors to the mesh the first time it is rendered.

The Unity Editor shows too many vertices or triangles (compared to what my 3D app says)

This is correct. What you are looking at is the number of vertices/triangles actually being sent to the GPU for **rendering**. In addition to the case where the material requires them to be sent twice, other things like hard-normals and non-contiguous UVs increase vertex/triangle counts significantly compared to what a modeling app tells you. Triangles need to be contiguous in both 3D and UV space to form a strip, so when you have UV seams, degenerate triangles have to be made to form strips - this bumps up the count.

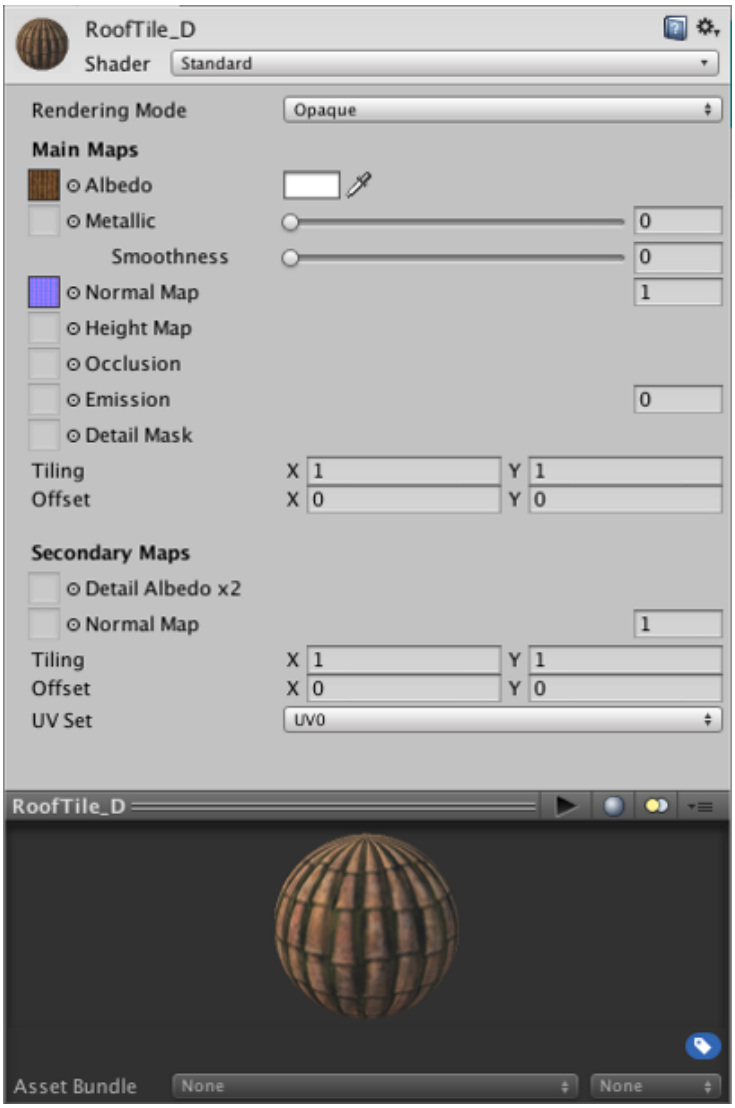
2018–04–25 Page amended with limited editorial review

Material

Leave feedback

SWITCH TO SCRIPTING

Materials are used in conjunction with Mesh Renderers, Particle Systems and other **rendering components** used in Unity. They play an essential part in defining how your object is displayed.



A typical Material inspector

Properties

The properties that a Material's **inspector** displays are determined by the **Shader** that the Material uses. A shader is a specialised kind of graphical program that determines how texture and lighting information are combined to generate the **pixels** of the rendered object onscreen. See the manual section about Shaders for in-depth information about how they are used in a Unity project.

Mesh Filter

Leave feedback

SWITCH TO SCRIPTING

The **Mesh Filter** takes a mesh from your **assets** and passes it to the Mesh Renderer for **rendering** on the screen.

Properties

Property: Function:

Mesh Reference to a mesh that will be rendered. The **Mesh** is located inside your Assets Directory.

Details

When importing mesh assets, Unity automatically creates a Skinned Mesh Renderer if the mesh is skinned, or a Mesh Filter along with a Mesh Renderer, if it is not.

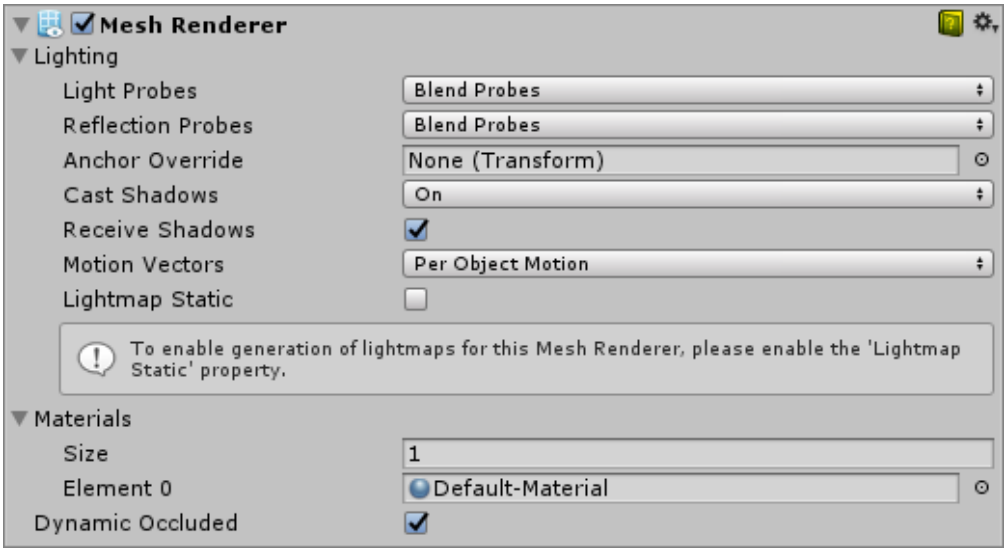
To see the Mesh in your **scene**, add a Mesh Renderer to the **GameObject**. It should be added automatically, but you will have to manually re-add it if you remove it from your object. If the Mesh Renderer is not present, the Mesh will still exist in your scene (and computer memory) but it will not be drawn.

Mesh Renderer

Leave feedback

SWITCH TO SCRIPTING

The **Mesh** Renderer takes the geometry from the Mesh Filter and renders it at the position defined by the **GameObject**'s Transform **component**.



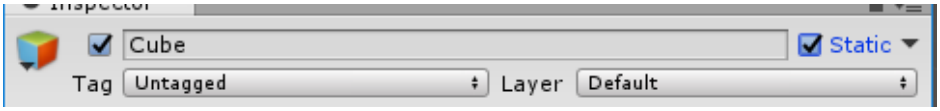
The **Mesh Renderer** GameObject Component as displayed in the **Inspector** window

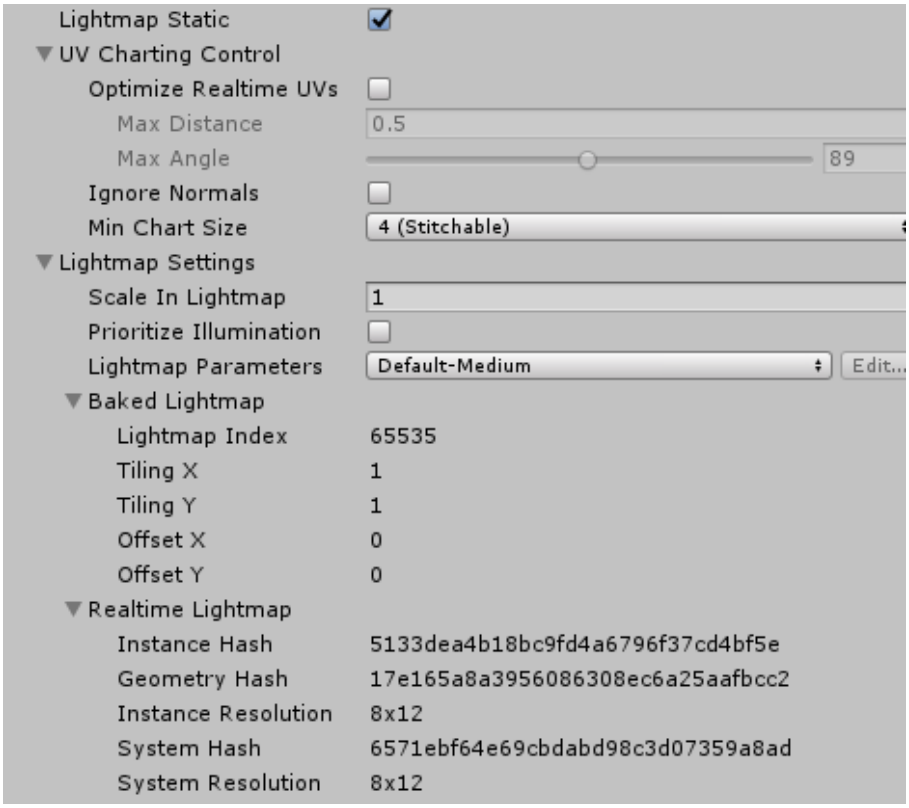
Properties

Property:	Function:
Light Probes	Probe-based lighting **interpolation** mode.
Off:	The Renderer doesn't use any interpolated Light Probes.
Blend Probes	The Renderer uses one interpolated Light Probe. This is the default option.
Use Proxy Volume	The Renderer uses a 3D grid of interpolated Light Probes.
Reflection Probes	Specifies how the GameObject is affected by reflections in the **scene**. You cannot disable this property in deferred **rendering modes**.
Off	Reflection probes are disabled, **skybox** will be used for reflection.
Blend Probes	Reflection probes are enabled. Blending occurs only between probes, useful in indoor environments. The renderer will use default reflection if there are no reflection probes nearby, but no blending between default reflection and probe will occur.
Blend Probes and Skybox	Reflection probes are enabled. Blending occurs between probes or probes and default reflection, useful for outdoor environments.
Simple	Reflection probes are enabled, but no blending will occur between probes when there are two overlapping volumes.
Anchor Override	A Transform used to determine the interpolation position when the Light Probe or Reflection Probe systems are used.

Property:	Function:
Cast Shadows	
On	The Mesh will cast a shadow when a shadow-casting Light shines on it
Off	The Mesh will not cast shadows
Two Sided	Two sided shadows are cast from either side of the Mesh. Two sided shadows are not supported by **Enlighten** or Progressive **Lightmapper**.
Shadows Only	Shadows from the Mesh will be visible, but not the Mesh itself
Receive Shadows	Enable this check box to make the Mesh display any shadows that are cast upon it. Review Shadows is only supported when using Progressive Lightmapper
Motion Vectors	If enabled, the line has motion vectors rendered into the **Camera** motion vector Texture. See Renderer.motionVectorGenerationMode in the Scripting API reference documentation to learn more.
Lightmap Static	Enable this check box to indicate to Unity that the GameObject's location is fixed and it will participate in Global Illumination computations. If an GameObject is not marked as **Lightmap** Static then it can still be lit using Light Probes.
Materials	A list of Materials to render the model with.
Dynamic Occluded	Enable this check box to indicate to Unity that **occlusion culling** should be performed for this GameObject even if it is not marked as static.

Tick the lightmap static checkbox to display MeshRenderer Lightmap information in the Inspector(see also the **static** checkbox of the GameObject).





UV Charting Control

Property:	Function:
Optimize Realtime Uvs	Specifies whether the authored Mesh UVs are optimized for Realtime Global Illumination or not. When enabled, the authored UVs are merged, scaled and packed for optimisation purposes. When disabled, the authored UVs will be scaled and packed, but not merged. Note that the optimization will sometimes make misjudgements about discontinuities in the original UV mapping. For example, an intentionally sharp edge may be misinterpreted as a continuous surface.
Max Distance	Specifies the maximum worldspace distance to be used for UV chart simplification. If charts are within this distance they will be simplified.
Max Angle	Specifies the maximum angle in degrees between faces sharing a UV edge. If the angle between the faces is below this value, the UV charts will be simplified.
Ignore normal	Check this box to prevent the UV charts from being split during the precompute process for Realtime Global Illumination lighting.
Min chart size	Specifies the minimum texel size used for a UV chart. If stitching is required a value of 4 will create a chart of 4x4 texels to store lighting and directionality. If stitching is not required, a value of 2 will reduce the texel density and provide better lighting build times and game performance.

Lightmap settings

Property:	Function:
Scale in Lightmap	This value specifies the relative size of the GameObject's UVs within a lightmap. A value of 0 will result in the GameObject not being lightmapped, but it will still contribute to lighting other GameObjects in the scene. A value greater than 1.0 increases the number of **pixels** (ie, the lightmap resolution) used for this GameObject while a value less than 1.0 decreases it. You can use this property to optimise lightmaps so that important and detailed areas are more accurately lit. For example: an isolated building with flat, dark walls will use a low lightmap scale (less than 1.0) while a collection of colourful motorcycles displayed close together warrant a high scale value.
Prioritize illumination	Check this box to tell Unity to always include this GameObject in lighting calculations. Useful for GameObjects that are strongly emissive to make sure that other GameObjects will be illuminated by this GameObject.
Lightmap Parameters	Allows you to choose or create a set of Lightmap Parameters for the this GameObject.

Details

Meshes imported from 3D packages can use multiple Materials. All the Materials used by a Mesh Renderer are held in the **Materials** list. Each sub-Mesh uses one Material from the Materials list. If there are more Materials assigned to the Mesh Renderer than there are sub-Meshes in the Mesh, the first sub-Mesh is rendered with each of the remaining Materials, one on top of the next. This allows you to set up multi-pass **rendering** on that sub-Mesh - but note that this can impact the performance at run time. Also note that fully opaque Materials, simply overwrite the previous layers, causing a decrease in performance with no advantage.

A Mesh can receive light from the Light Probe system and reflections from the Reflection Probe system depending on the settings of the **Use Light Probes** and **Use Reflection Probes** options. For both types of probe, a single point is used as the Mesh's notional position probe interpolation. By default, this is the centre of the Mesh's bounding box, but you can change this by dragging a Transform to the **Anchor Override** property (the Anchor Override affects both types of probe).

It may be useful to set the anchor in cases where a GameObject contains two adjoining Meshes; since each Mesh has a separate bounding box, the two are lit discontinuously at the join by default. However, if you set both Meshes to use the same anchor point, then they are consistently lit. By default, a probe-lit Renderer receives lighting from a single Light Probe that is interpolated from the surrounding Light Probes in the Scene. Because of this, GameObjects have constant ambient lighting across the surface. It has a rotational gradient because it is using spherical harmonics, but it lacks a spatial gradient. This is more noticeable on larger GameObjects or **Particle Systems**. The lighting across the GameObject matches the lighting at the anchor point, and if the GameObject straddles a lighting gradient, parts of the GameObject look incorrect.

To alleviate this behavior, set the **Light Probes** property to **Use Proxy Volume**, with an additional Light Probe Proxy Volume component. This generates a 3D grid of interpolated Light Probes inside a **bounding volume** where the resolution of the grid can be user-specified. The spherical harmonics coefficients of the interpolated Light Probes are updated into 3D Textures, which are sampled at render time to compute the contribution to the diffuse ambient lighting. This adds a spatial gradient to probe-lit GameObjects.

2018–09–21 Page amended with limited editorial review

Updated supported lighting backends for Two Sided shadows and Recieve shadows.

Mesh Renderer UI updated in 5.6

Skinned Mesh Renderer

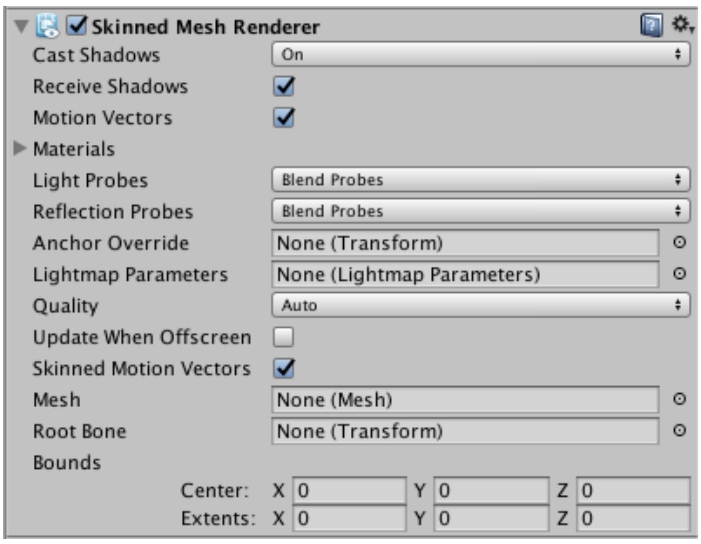
Leave feedback

SWITCH TO SCRIPTING

Unity uses the **Skinned Mesh Renderer component** to render **Bone** animations, where the shape of the Mesh is deformed by predefined animation sequences. This technique is useful for characters and other objects whose **joints** bend (as opposed to a machine where joints are more like hinges).

A Skinned **Mesh Renderer** is automatically added to any Mesh that needs it at import time.

Properties



Property:	Function:
Cast Shadows	If enabled, the Mesh will cast shadows when a suitable Light shines on it.
Receive Shadows	If enabled, the Mesh will show shadows that are cast upon it by other objects.
Motion Vectors	If enabled, the line has motion vectors rendered into the **Camera** motion vector Texture. See Renderer.motionVectors in the Scripting API reference documentation to learn more.
Materials	The list of **Materials** the Mesh will be rendered with.
Light Probes	Probe-based lighting **interpolation** mode.
Reflection Probes	If enabled, and if Reflection Probes are present in the **scene**, a reflection Texture will be picked for this **GameObject** and set as a built-in **Shader** uniform variable.
Anchor Override	A Transform used to determine the interpolation position when the Light Probe or Reflection Probe systems are used.
Quality	Define the maximum number of bones used per vertex while **skinning**. The higher the number of bones, the higher the quality of the Renderer. Set the **Quality** to **Auto** to use the **Blend Weights** value from the Quality Settings.
Skinned Motion Vectors	If enabled, the Mesh skinning data will be double buffered so that skinned motion can be interpolated and placed into the motion vector Texture. This has a GPU memory overhead, but leads to more correct motion vectors.
Update When Offscreen	If enabled, the Skinned Mesh will be updated even when it can't be seen by any Camera. If disabled, the animations themselves will also stop running when the GameObject is off-screen.
Mesh	Use this to define the Mesh used by this Renderer.

Property:	Function:
Root Bone	Use this to define the bone that is the "root" of the animation (that is, the bone relative to which all the others move).
Bounds	The **bounding volume** that is used to determine when the Mesh is offscreen. The bounds are pre-calculated on import from the Mesh and animations in the **model file**, and are displayed as a wireframe around the model in the **Scene View**.

Details

Bones are invisible objects inside a skinned Mesh that affect the way the Mesh is deformed during animation. The basic idea is that the bones are joined together to form a hierarchical "skeleton", and the animation is defined by rotating the joints of the skeleton to make it move. Each bone is attached to some of the vertices of the surrounding Mesh. When the animation is played, the vertices move with the bone or bones they are connected to, so the "skin" follows the movement of the skeleton. At a simple joint (for example, an elbow), the Mesh vertices are affected by both of the bones that meet there, and the Mesh will stretch and rotate realistically as the joint bends. In more complex situations, more than two bones will affect a particular area of the Mesh, resulting in more subtle movements.

Although a skinned Mesh is most commonly used with predefined animations, it is also possible to attach Rigidbody components to each bone in a skeleton to put it under the control of the **Physics engine**. This is typically used to create the "ragdoll" effect, where a character's limbs flail after being thrown or struck by an explosion.

Quality

Unity can skin every vertex with one, two or four bones. Using four bones gives the best results but this comes with a higher processing overhead. Games commonly use two bone weights, which is a good compromise between visual quality and performance.

If the **Quality** is set to **Auto**, the **Blend Weights** value from the Quality Settings are used. This allows end-users to choose the quality setting themselves and get the desired balance of animation quality and framerate.

Update When Offscreen

By default, skinned Meshes that are not visible to any camera are not updated. The skinning is not updated until the Mesh comes back on screen. This is done to save system resources.

The object's visibility is determined from the Mesh's **Bounds** (that is, the entire bounding volume must be outside the view of any active Camera). However, the true bounding volume of an animated Mesh can change as the animation plays (for example, the bounding volume will get taller if the character raises their hand in the air). Unity takes all attached animations into account when calculating the maximum bounding volume, but there are cases when the bounds can't be calculated to anticipate every possible use case.

Each of the following example situations become a problem when they push bones or vertices out of the pre-calculated bounding volume:

animations added at run-time
additive animations
procedurally changing the positions of bones from a script
using **vertex shaders** which can push vertices outside the pre-calculated bounds
using ragdolls

In those cases, there are two solutions:

Modify the Bounds to match the potential bounding volume of your Mesh
Enable **Update When Offscreen** to skin and render the skinned Mesh all the time

You should usually use the first option, because it is better for performance. However, the second option is preferable if performance is not a major concern, or if you can't predict the size of your bounding volume (for example, when using ragdolls).

In order to make Skinned Meshes work better with ragdolls, Unity will automatically remap the Skinned Mesh Renderer to the root bone on import. However Unity only does this if there is a single Skinned Mesh Renderer in the model file. This means that if you can't attach all Skinned Mesh Renderers to the root bone or a child, and you use ragdolls, you should turn off this optimization.

Importing skinned Meshes

Currently, skinned Meshes can be imported from:

Autodesk® Maya®
Cinema4D
Autodesk® 3ds Max®
Blender
Cheetah 3D
XSI
Any tool that supports the FBX format

On mobile devices, Unity handles skinning on the CPU with hand-coded NEON/VFP assembly. A limitation here is that normals and tangents are not normalized, so if you are writing your own Shaders, you should handle the normalization youself. However, if you are using Surface Shaders, Unity automatically handles the normalization.

Note: Optimized Meshes sort bones differently from non-optimized Meshes, resulting in potentially significant animation problems. This is because non-optimized Meshes rely on bone order to animate, while Optimized Meshes use the bone names and do not rely on bone order.

If you simply import the FBX file and use it, Unity will take care of the order of the transforms.

For advanced users, if you want to change `SkinnedMeshRenderer.sharedMesh`:

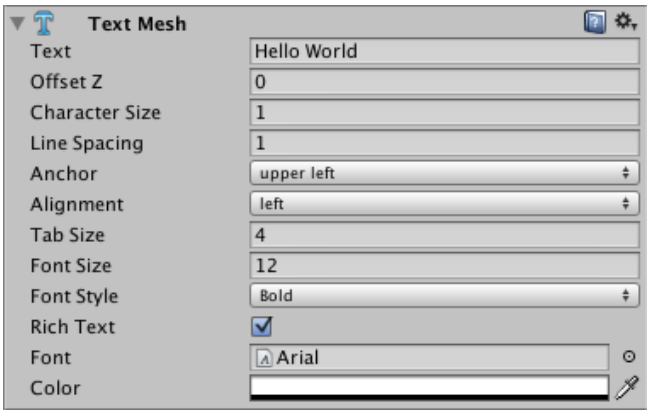
In 'non-optimized' mode, you need to make sure that the `SkinnedMeshRenderer.bones` matches `SkinnedMeshRenderer.sharedMesh` in a strict way. The referenced Transforms should be there in the correct order.
In optimized mode, it's much simpler; the **rendering** works as long as the **avatar** has the referenced bones. In this case, `SkinnedMeshRenderer.bones` is always empty.

Text Mesh

Leave feedback

SWITCH TO SCRIPTING

The **Text Mesh** generates 3D geometry that displays text strings.



You can create a new Text Mesh from **Component > Mesh > Text Mesh**.

Properties

Property:	*Function:*
Text	The text that will be rendered
Offset Z	How far should the text be offset from the transform.position.z when drawing
Character Size	The size of each character (This scales the whole text.)
Line Spacing	How much space will be in-between lines of text.
Anchor	Which point of the text shares the position of the Transform.
Alignment	How lines of text are aligned (Left, Right, Center).
Tab Size	How much space will be inserted for a tab '\t' character. This is a multiplum of the 'spacebar' character offset.
Font Size	The size of the font. This can override the size of a dynamic font.
Font Style	The **rendering** style of the font. The font needs to be marked as dynamic.
Rich Text	When selected this will enable tag processing when the text is rendered.
Font	The TrueType Font to use when rendering the text.
Color	The global color to use when rendering the text.

Details

Text Meshes can be used for rendering road signs, graffiti **etc**. The Text **Mesh** places text in the 3D **scene**. To make generic 2D text for GUIs, use a GUI Text **component** instead.

Follow these steps to create a Text Mesh with a custom Font:

Import a font by dragging a TrueType Font - a **.ttf** file - from the Explorer (Windows) or Finder (OS X) into the **Project View**.
Select the imported font in the Project View.
Choose **GameObject > Create Other > 3D Text**. You have now created a text mesh with your custom TrueType Font.
You can scale the text and move it around using the **Scene View's Transform** controls.
Note: If you want to change the font for a Text Mesh, need to set the component's font property and also set the texture of the font material to the correct font texture. This texture can be located using the font **asset**'s foldout. If you forget to set the texture then the text in the mesh will appear blocky and misaligned.

Hints

You can download free TrueType Fonts from 1001freefonts.com (download the Windows fonts since they contain TrueType Fonts).
If you are scripting the **Text** property, you can add line breaks by inserting the escape character "\n" in your strings.
Text meshes can be styled using simple mark-up. See the Styled Text page for more details.
Fonts in Unity are rendered by first rendering the font glyphs to a texture map. If the font size is set too small, these font textures will appear blocky. Since TextMesh assets are rendered using **quads**, it's possible that if the size of the TextMesh and font texture differ the TextMesh will look wrong.

Text Asset

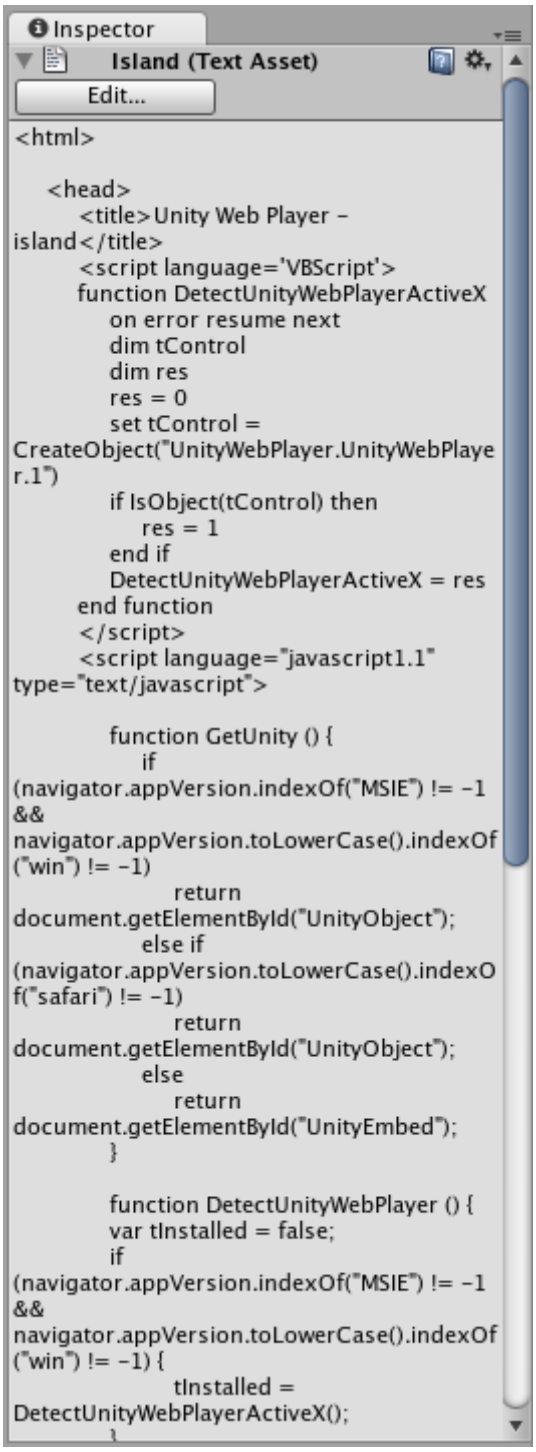
Leave feedback

SWITCH TO SCRIPTING

Text Assets are a format for imported text files. When you drop a text file into your Project Folder, it will be converted to a Text **Asset**. The supported text formats are:

.txt
.html
.htm
.xml
.bytes
.json
.csv
.yaml
.fnt

Note that script files are also considered text assets for the purposes of using the AssetDatabase.FindAssets function, so they will also be included in the list of results when this function is used with the "t:TextAsset" filter.



The Text Asset Inspector

Properties

Property: Function:

Text The full text of the asset as a single string.

Details

The Text Asset is a very specialized use case. It is extremely useful for getting text from different text files into your game while you are building it. You can write up a simple .txt file and bring the text into your game very easily. It is not intended for text file generation at runtime. For that you will need to use traditional Input/Output programming techniques to read and write external files.

Consider the following scenario. You are making a traditional text-heavy adventure game. For production simplicity, you want to break up all the text in the game into the different rooms. In this case you would make one text file that contains all the text that will be used in one room. From there it is easy to make a reference to the correct Text Asset for the room you enter. Then with some customized parsing logic, you can manage a large amount of text very easily.

Binary data

A special feature of the text asset is that it can be used to store binary data. By giving a file the extension **.bytes** it can be loaded as a text asset and the data can be accessed through the **bytes** property.

For example put a jpeg file into the Resources folder and change the extension to **.bytes**, then use the following script code to read the data at runtime:

```
//Load texture from disk
TextAsset bindata= Resources.Load("Texture") as TextAsset;
Texture2D tex = new Texture2D(1,1);
tex.LoadImage(bindata.bytes);
```

Please notice that files with the **.txt** and **.bytes** extension will be treated as text and binary files, respectively. Do not attempt to store a binary file using the **.txt** extension, as this will create unexpected behaviour when attempting to read data from it.

Hints

Text Assets are serialized like all other assets in a build. There is no physical text file included when you publish your game.
Text Assets are not intended to be used for text file generation at runtime.

Font

Leave feedback

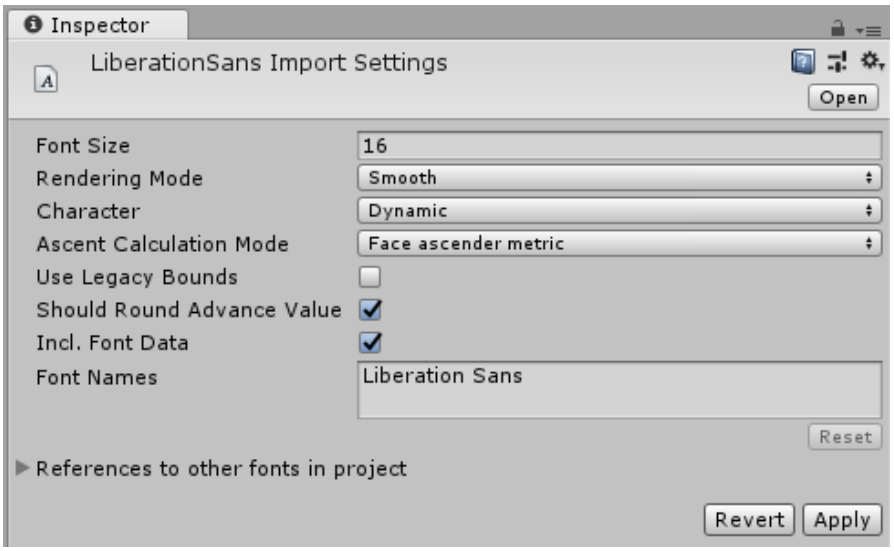
SWITCH TO SCRIPTING

Fonts can be created or imported for use in either the GUI Text or the Text Mesh **Components**.

Importing Font files

To add a font to your project you need to place the font file in your **Assets** folder. Unity will then automatically import it. Supported Font formats are TrueType Fonts (.ttf files) and OpenType Fonts (.otf files).

To change the **Size** of the font, highlight it in the **Project View** and you have a number of options in the **Import Settings** in the **Inspector**.



Import Settings for a font

Property:	*Function:*
Font Size	The size of the font, based on the sizes set in any word processor.
Rendering mode	The font **rendering** mode, which tells Unity how to apply smoothing to the glyphs.
Character	The character set of the font to import into the font texture Setting this mode to Dynamic causes Unity to embed the font data itself and render font glyphs at runtime (see below).

Import Settings specific to dynamic fonts

Property:	*Function:*
Include Font Data	This setting controls the packaging of the font when used with Dynamic font property. When selected the TTF is included in the output of the build. When not selected it is assumed that the end user will have the font already installed on their machine. Note that fonts are subject to copyright and you should only include fonts that you have licensed or created for yourself.
Font Names	A list of fallback fonts to use when fonts or characters are not available (see below).

After you import the font, you can expand the font in Project View to see that it has auto-generated some assets. Two assets are created during import: “font material” and “font texture”. Unlike many applications you might be familiar with, fonts in Unity are converted into textures, and the glyphs that you display are rendered using textured **quads**. Adjusting the font size effectively changes how many **pixels** are used for each glyph in this generated texture. Text **Mesh** assets are 3d geometry textured with these auto-generated font textures. You will want to vary the size of the font to make these assets look crisp.

Dynamic fonts

When you set the **Characters** drop-down in the Import Settings to **Dynamic**, Unity will not pre-generate a texture with all font characters. Instead, it will use the FreeType font rendering engine to create the texture on the fly. This has the advantage that it can save in download size and texture memory, especially when you are using a font which is commonly included in user systems, so you don't have to include the font data, or when you need to support asian languages or large font sizes (which would make the font textures very large using normal font textures).

When Unity tries to render text with a dynamic font, but it cannot find the font (because **Include Font Data** was not selected, and the font is not installed on the user machine), or the font does not include the requested glyph (like when trying to render text in east Asian scripts using a latin font, or when using styled bold/italic text), then it will try each of the fonts listed in the **Font Names** field, to see if it can find a font matching the font name in the project (with font data included) or installed on the user machine which has the requested glyph. If none of the listed fallback fonts are present and have the requested glyph, Unity will fall back to a hard-coded global list of fallback fonts, which contains various international fonts commonly installed on the current runtime platform.

Note that some target platforms (WebGL, some consoles) do not have OS default fonts Unity can access for rendering text. For those platforms, **Include Font Data** will be ignored, and font data will always be included. All fonts to be used as fallbacks must be included in the project, so if you need to render international text or bold/italic versions of a font, you need to add a font file which has the required characters to the project, and set up that font in the **Font Names** list of other fonts which should use it as fallbacks. If the fonts are set up correctly, the fallback fonts will be listed in the Font Importer inspector, as *References to other fonts in project*.

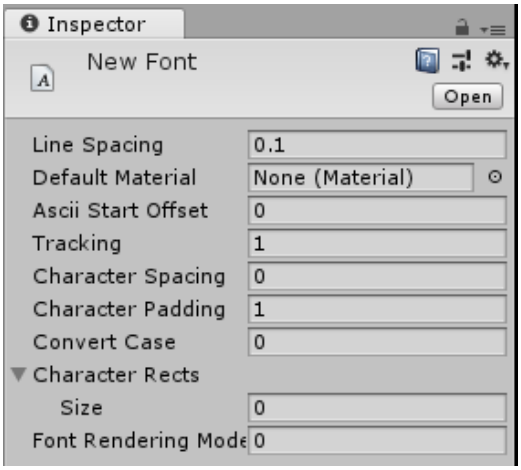
Default font asset

The default font asset is a dynamic font which is set up to use Arial. If Unity can't find the Arial font on your computer (for example, if you don't have it installed), it will fall back to a font bundled with Unity called Liberation Sans.

Liberation Sans looks like Arial, but it does not include bold or italic font styles, and only has a basic Latin character set - so styled text or non-latin characters may fall back to other fonts or fail to render. It does however have a license which allows it to be included in player builds.

Custom fonts

To create a custom font select 'Create->custom font' from the project window. This will add a custom font asset to your project library.

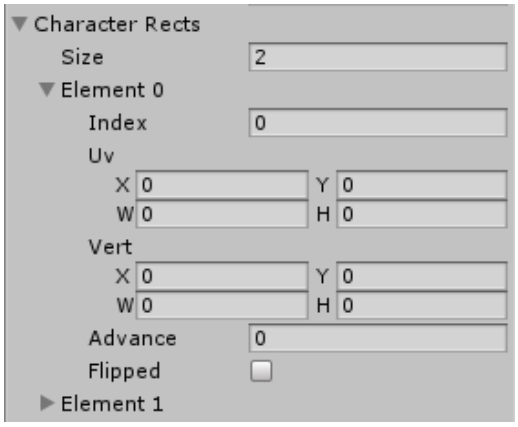


The **Ascii Start Offset** field is a decimal that defines the Ascii index you would like to begin your **Character Rects** index from. For example, if your Ascii Start Offset is set to 0 then the capital letter A will be at index 65 but if the Ascii Start Offset is set to 65 then the letter A will be at index 0. You can consult the Ascii Table here but you should bear in mind that custom font uses the decimal ascii numbering system.

Tracking can be set to modify how close each character will be to the next character on the same line and **Line spacing** can be set to define how close each line will be to the next.

To create a font material you will need to import your font as a texture then apply that texture to a material, then drag your font material onto the **Default Material** section.

The **Character Rects** section is where each character of your font is defined.



The **Size** field is for defining how many characters are in your font.

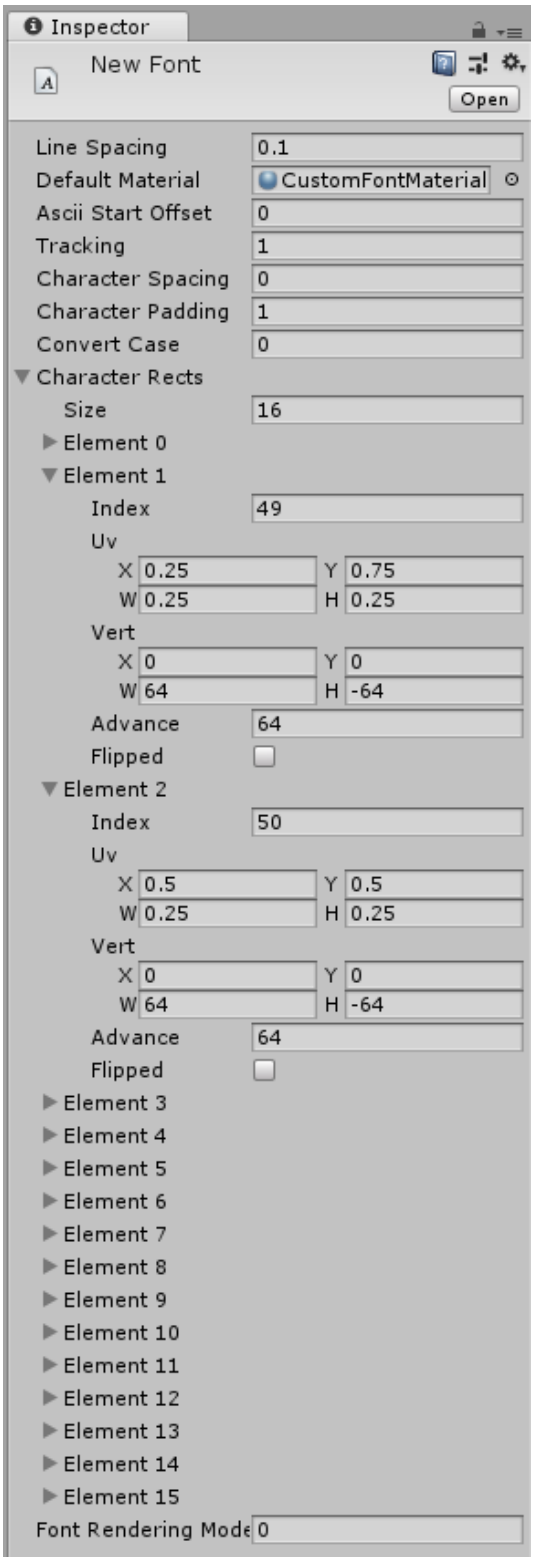
Within each **Element** there is an index field for the ascii index of the character. This will be an integer that represents the character in this element.

To work out the **UV** values you need to figure out how your characters are positioned on a scale of 0 to 1. You divide 1 by the number of characters on a dimension. For example if you have a font and the image dimensions on it are 256x128, 4 characters across, 2 down (so 64x64), then UV width will be 0.25 and UV height will be 0.5.

For UV X and Y, it's just a matter of deciding which character you want and multiplying the width or height value times the column/row of the letter.

Vert size is based on the pixel size of the characters e.g. your characters are each 128x128, putting 128 and –128 into the Vert Width and Height will give properly proportioned letters. Vert Y must be negative.

Advance will be the desired horizontal distance from the origin of this character to the origin of the next character in pixels. It is multiplied by **Tracking** when calculating the actual distance.



Example of custom font inspector with values

Unicode support

Unity has full unicode support. Unicode text allows you to display German, French, Danish or Japanese characters that are usually not supported in an ASCII character set. You can also enter a lot of different special purpose characters like arrow signs or the option key sign, if your font supports it.

To use unicode characters, choose either **Unicode** or **Dynamic** from the **Characters** drop-down in the Import Settings. You can now display unicode characters with this font. If you are using a **GUIText** or **Text Mesh**, you can enter unicode characters into the Component's **Text** field in the Inspector.

You can also use unicode characters if you want to set the displayed text from scripting. The C# compiler fully supports Unicode based **scripts**. You have to save your **scripts** with UTF–16 encoding. Now you can add unicode characters to a string in your script and they will display as expected in **UnityGUI**, a GUIText, or a Text Mesh.

Note that surrogate pairs are not supported.

Changing Font Color

There are different ways to change the color of your displayed font, depending on how the font is used.

GUIText and Text Mesh

If you are using a GUIText or a Text Mesh, you can change its color by using a custom **Material** for the font. In the Project View, click on **Create > Material**, and select and set up the newly created Material in the **Inspector**. Make sure you assign the texture from the font asset to the material. If you use the built-in **GUI/Text Shader** shader for the font material, you can choose the color in the **Text Color** property of the material.

UnityGUI

If you are using UnityGUI scripting to display your font, you have much more control over the font's color under different circumstances. To change the font's color, you create a **GUISkin** from **Assets > Create > GUI Skin**, and define the color for the specific control state, e.g. **Label > Normal > Text Color**. For more details, please read the GUI Skin page.

Hints

To display an imported font select the font and choose **GameObject > Create Other > 3D Text**.
You can reduce the generated texture size for fonts by using only lower or upper case characters.

Texture Components

Leave feedback

This group contains all **Components** that have to do with Textures.

Textures

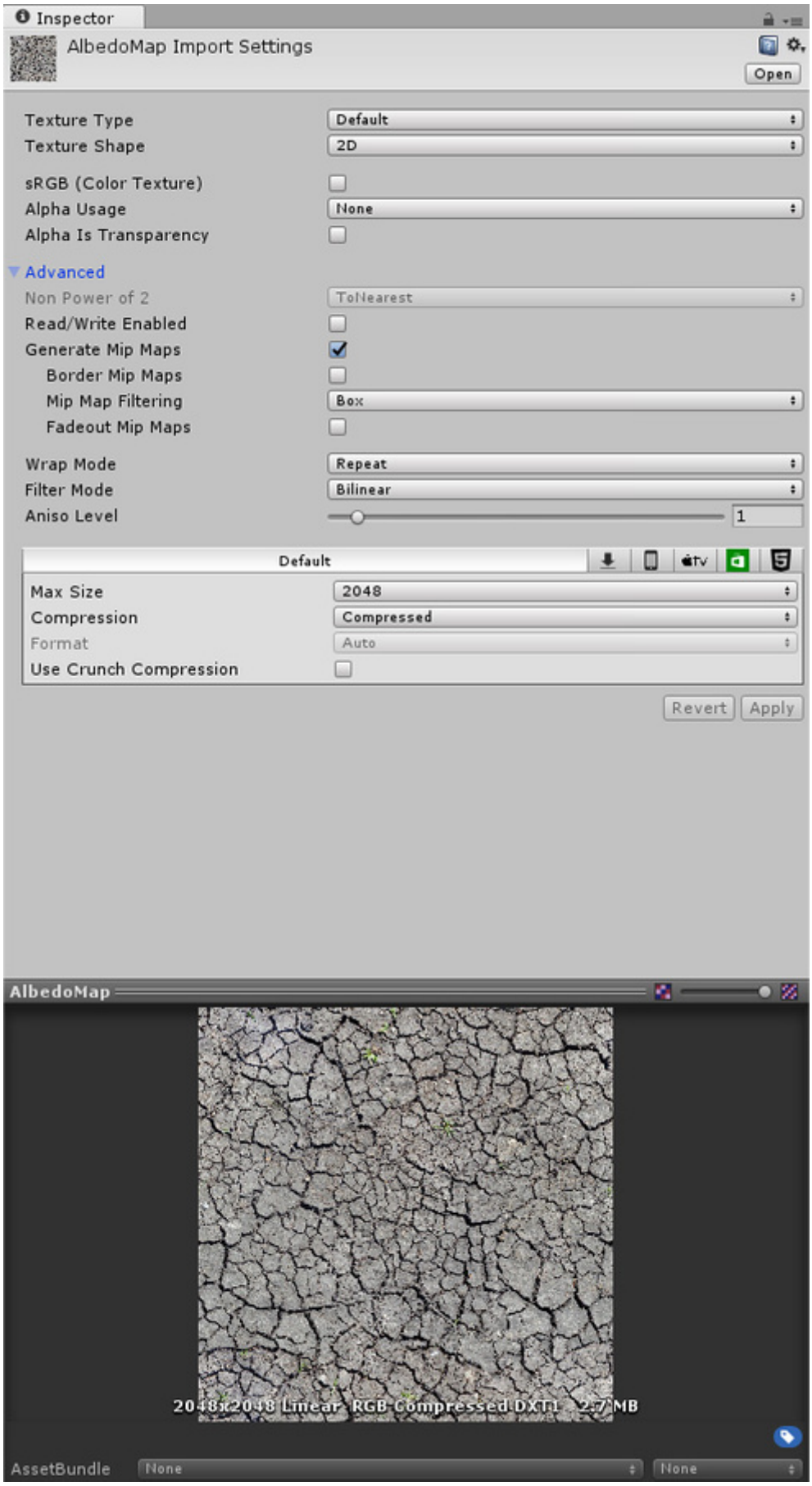
Leave feedback

SWITCH TO SCRIPTING

Textures are image or movie files that lay over or wrap around your **GameObjects** to give them a visual effect. This page details the properties you need to manage for your Textures.

Unity recognises any image or movie file in a 3D project's *Assets* folder as a Texture (in 2D projects, they are saved as Sprites). As long as the image meets the size requirements specified below, it is imported and optimized for game use (although any Shaders you use for your GameObjects have specific Texture requirements). This extends to multi-layer Photoshop or TIFF files, which are automatically flattened on import so that there is no size penalty for your game. This flattening happens internally to Unity, not to the PSD file itself, and is optional, so you can continue to save and import your PSD files with layers intact.

Properties



The Texture Inspector window

The **Inspector** window is split into two sections: the **Texture Importer** above, and the **Preview** below.

Texture Importer

The Texture Importer defines how images are imported from your project's *Assets* folder into the Unity Editor. To access the Texture Importer, select the image file in the Project window. The Texture Importer opens in the Inspector window.

Note that some of the less commonly used properties are hidden by default. Click **Advanced** in the Inspector window to view these.

The first property in the Texture Importer is the **Texture Type**. Use this to select the type of Texture you want to create from the source image file. See documentation on Texture types for more information on each type.

Property:	Function:
Texture Type	Use this to define what your Texture is to be used for. The other properties in the Texture Importer change depending on which one you choose.
Default	This is the most common setting used for all Textures. It provides access to most of the properties for Texture importing.
Normal Map	Select this to turn the color channels into a format suitable for real-time normal mapping. See Importing Textures for more information on normal mapping.
Editor GUI	Select this if you are using the Texture on any HUD or GUI controls.
Sprite (2D and UI)	Select this if you are using the Texture in a 2D game as a Sprite.
Cursor	Select this if you are using the Texture as a custom cursor.
Cookie	Select this to set your Texture up with the basic parameters used for the Cookies of your **Scene**'s Lights.
Lightmap	Select this if you are using the Texture as a Lightmap. This option enables encoding into a specific format (RGBM or dLDR, depending on the platform) and a post-processing step on Texture data (a push-pull dilation pass).
Single Channel	Select this if you only need one channel in the Texture.

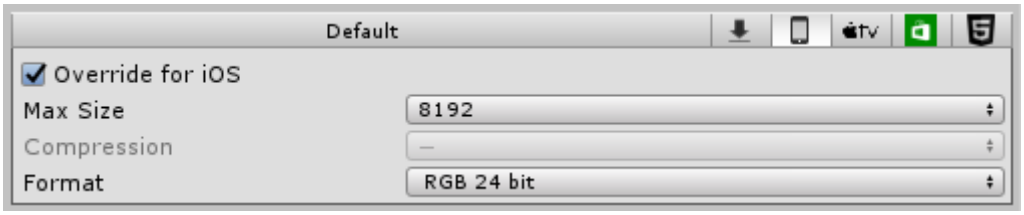
The second property in the Texture Importer is the **Texture Shape**. Use this to select and define the shape and structure of the Texture.

Property:	Function:
Texture Shape	Use this to define the shape of the Texture. This is set to 2D by default.
2D	This is the most common setting for all Textures; it defines the image file as a 2D Texture. These are used to map textures to 3D meshes and GUI elements, among other project elements.
Cube	This defines the Texture as a **cubemap**. You could use this for Skyboxes or **Reflection Probes**, for example. Selecting **Cube** displays different mapping options.
Mapping	This setting is only available when **Texture Shape** is set to **Cube**. Use **Mapping** to specify how the Texture is projected onto your GameObject. This is set to **Auto** by default.
Auto	Unity tries to automatically work out the layout from the Texture information.

Property:	Function:
6 Frames Layout (Cubic Environment)	The Texture contains six images arranged in one of the standard cubemap layouts: cross, or sequence (+x -x +y -y +z -z). The images can be orientated either horizontally or vertically.
Latitude Longitude (Cylindrical)	Maps the Texture to a 2D Latitude-Longitude representation.
Mirrored Ball (Sphere Mapped)	Maps the Texture to a sphere-like cubemap.
Convolution Type	Choose the type of pre-convolution (that is, filtering) that you want to use for this texture. The result of pre-convolution is stored in mips. This is set to **None** by default.
None	The Texture has no pre-convolution (no filtering).
Specular (Glossy Reflection)	Select this to use cubemaps as Reflection Probes. The Texture mip maps are pre-convoluted (filtered) with the engine BRDF. (See Wikipedia's page on Bidirectional reflectance distribution function for more information.)
Diffuse (Irradiance)	The Texture is convoluted (filtered) to represent irradiance. This is useful if you use the cubemap as a **Light Probe**.
Fixup Edge Seams	This option is only available with the **None** or **Diffuse** convolution (filter). Use this on low-end platforms as a work-around for filtering limitations, such as cubemaps incorrectly filtered between faces.

Platform-specific overrides

The Texture Inspector window has a **Platform-specific overrides** panel.



Platform-specific overrides panel

When building for different platforms, you need to think about the resolution, the file size with associated memory size requirements, the **pixel** dimensions, and the quality of your Textures for each target platform. Use the **Platform-specific overrides** panel to set default options (using **Default**), and then override them for a specific platform using the buttons along the top of the panel.

Property:	Function:
Max Size	The maximum imported Texture dimensions in pixels. Artists often prefer to work with huge dimension-size Textures; use **Max Size** to scale the Texture down to a suitable dimension-size.
Compression	Choose the compression type for the Texture. This parameter helps the system choose the right compression format for a Texture. Depending on the platform and the availability of compression formats, different settings might end up with the same internal format (for example, **Low Quality Compression** has an effect on mobile platforms, but not on desktop platforms).

Property:	Function:
None	The Texture is not compressed.
Low Quality	The Texture is compressed in a low-quality format. This results in a lower memory usage compared with **Normal Quality**.
Normal Quality	The Texture is compressed with a standard format.
High Quality	The Texture is compressed in a high-quality format. This results in a higher memory usage compared with **Normal Quality**.
Format	This bypasses the automatic system to specify what internal representation is used for the Texture. The list of available formats depends on the platform and Texture type. See documentation on Texture formats for platform-specific overrides for more information. **Note:** Even when a platform is not overridden, this option shows the format chosen by the automatic system. The Format property is only available when overriding for a specific platform, and not as a default setting.
Use crunch compression	Use crunch compression, if applicable. Crunch is a lossy compression format on top of DXT or **ETC Texture compression**. Textures are decompressed to DXT or **ETC** on the CPU and then uploaded on the GPU at runtime. Crunch compression helps the Texture use the lowest possible amount of space on disk and for downloads. Crunch Textures can take a long time to compress, but decompression at runtime is very fast.
Compressor Quality	When using Crunch Texture compression, use the slider to adjust the quality. A higher compression quality means larger Textures and longer compression times.

2017–09–18 Page amended with limited editorial review

Crunch compression format updated in 2017.3

Importing Textures

Leave feedback

This page offers details and tips on importing Textures using the Unity Editor Texture Importer. Scroll down or select an area you wish to learn about.

Supported file formats
HDR Textures
Texture sizes
UV mapping
Mip maps
Normal maps
Detail maps
Reflections (cubemaps)
Anisotropic filtering

Supported file formats

Unity can read the following file formats:

BMP
EXR
GIF
HDR
IFF
JPG
PICT
PNG
PSD
TGA
TIFF

Note that Unity can import multi-layer PSD and TIFF files; they are flattened automatically on import, but the layers are maintained in the **Assets** themselves. This means that you don't lose any of your work when using these file types natively. This is important, because it allows you to have just one copy of your Textures which you can use in different applications; Photoshop, your 3D modelling app, and in Unity.

HDR Textures

When importing from an EXR or **HDR** file containing **HDR** information, the Texture Importer automatically chooses the right HDR format for the output Texture. This format changes automatically depending on which platform you are building for.

Texture dimension sizes

Ideally, Texture dimension sizes should be powers of two on each side (that is, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 pixels (px), and so on). The Textures do not have to be square; that is the width can be different from height. Note that specific platforms may impose maximum Texture dimension sizes. For DirectX, the maximum Texture sizes for different feature levels are as follows:

Graphics APIs / Feature levels	Maximum 2D and **Cubemap** texture dimension size (px)
DX9 **Shader** Model 2 (PC GPUs before 2004) / OpenGL ES 2.0	2048
DX9 Shader Model 3 (PC GPUs before 2006) / Windows Phone DX11 9.3 level / OpenGL ES 3.0	4096
DX10 Shader Model 4 / GL3 (PC GPUs before 2007) / OpenGL ES 3.1	8192
DX11 Shader Model 5 / GL4 (PC GPUs since 2008)	16384

Notes:

The Texture Importer only allows you to choose dimension sizes up to 8K (that is 8192 x 8192 px).
Mali-Txxx GPUs (See Wikipedia) and OpenGL ES 3.1 (www.opengl.org) only support up to 4096px Texture dimension size for cubemaps.
It is possible to use NPOT (non-power of two) Texture sizes with Unity; however, NPOT Texture sizes generally take slightly more memory and might be slower for the GPU to sample, so it's better for performance to use power of two sizes whenever you can. If the platform or GPU does not support NPOT Texture sizes, Unity scales and pads the Texture up to the next power of two size. This process uses more memory and makes loading slower (especially on older mobile devices). In general, you should only use NPOT sizes for GUI purposes.

You can scale up NPOT Texture Assets at import time using the **Non Power of 2** option in the **Advanced** section of the Texture Importer.

UV mapping

When mapping a 2D Texture onto a 3D model, your 3D modelling application does a type of wrapping called UV mapping. Inside Unity, you can scale and move the Texture using Materials. Scaling normal and detail maps is especially useful.

Mip maps

Mip maps are lists of progressively smaller versions of an image, used to optimise performance on real-time 3D engines. Objects that are far away from the **Camera** use smaller Texture versions. Using mip maps uses 33% more memory, but not using them can result in a huge performance loss. You should always use mip maps for in-game Textures; the only exceptions are Textures that are made smaller (for example, GUI textures, Skybox, Cursors and Cookies). Mip maps are also essential for avoiding many forms of Texture aliasing and shimmering.

Normal maps

Normal maps are used by normal map Shaders to make low-polygon models look as if they contain more detail. Unity uses normal maps encoded as RGB images. You also have the option to generate a normal map from a grayscale height map image.

Detail maps

If you want to make a **Terrain**, you normally use your main Texture to show areas of **terrain** such as grass, rocks and sand. If your **terrain** is large, it may end up very blurry. Detail Textures hide this fact by fading in small details as your main Texture gets closer.

When drawing Detail Textures, a neutral gray is invisible, white makes the main Texture twice as bright, and black makes the main Texture completely black.

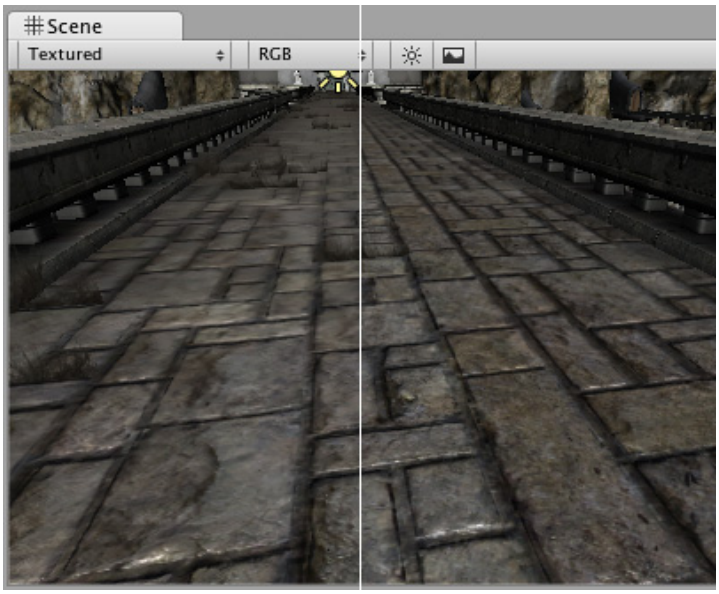
See documentation on Secondary Maps (Detail Maps) for more information.

Reflections (cubemaps)

To use a Texture for reflection maps (for example, in Reflection Probes or a cubemapped Skybox, set the **Texture Shape** to **Cube**. See documentation on Cubemap Textures for more information.

Anisotropic filtering

Anisotropic filtering increases Texture quality when viewed from a grazing angle. This **rendering** is resource-intensive on the graphics card. Increasing the level of Anisotropy is usually a good idea for ground and floor Textures. Use Quality Settings to force anisotropic filtering for all Textures or disable it completely.



Anisotropy used on the ground Texture {No anisotropy (left) | Maximum anisotropy (right)}

Texture Types

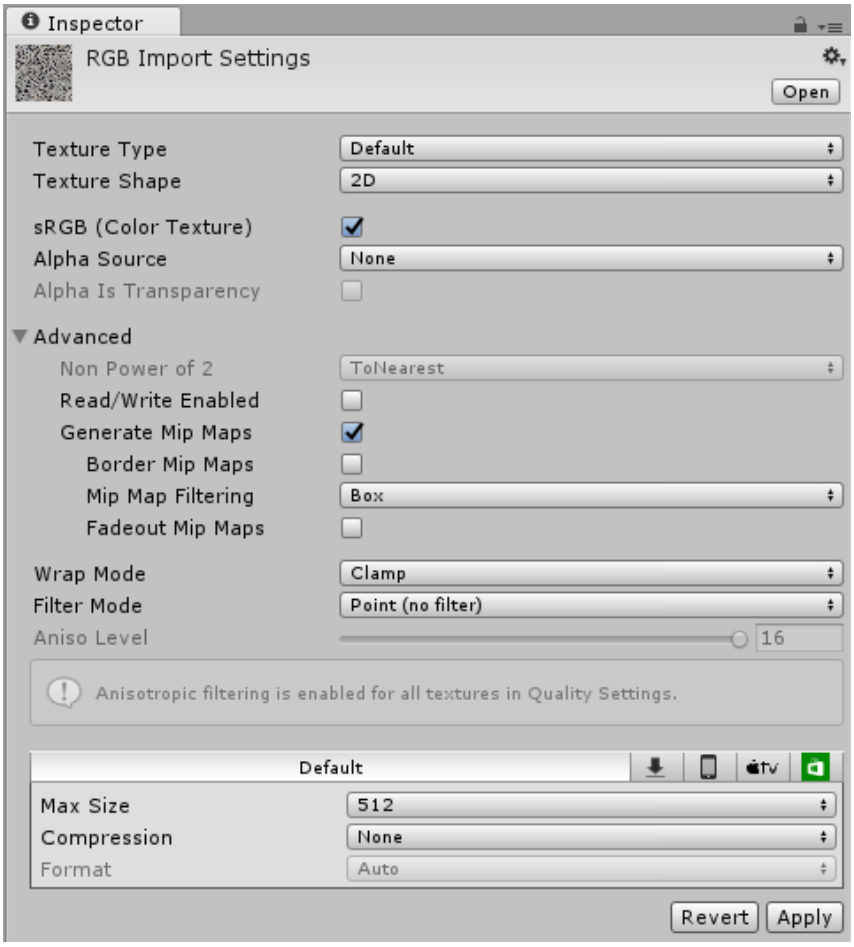
Leave feedback

You can import different Texture types into the Unity Editor via the Texture Importer.

Below are the properties available to configure the various Texture types in Unity in the Texture **Inspector** window. Scroll down or select from the list below to find details of the Texture type you wish to learn about.

Default
Normal Map
Editor GUI and Legacy
Sprite (2D and UI)
Cursor
Cookie
Lightmap
Single Channel

Texture type: Default



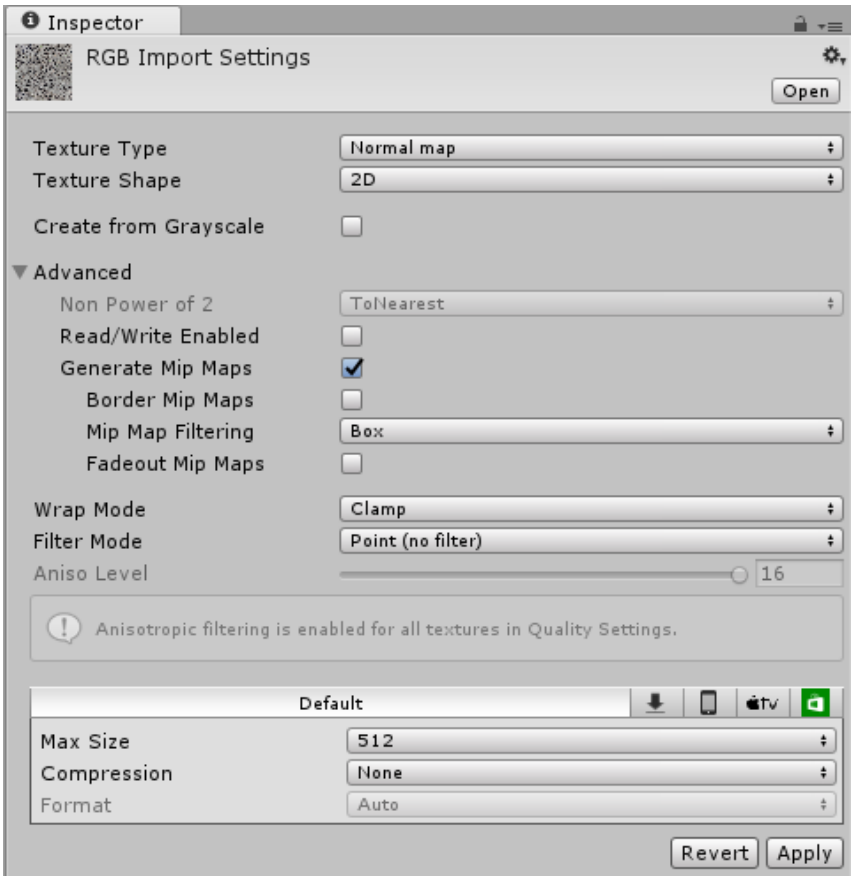
Texture Inspector window - **Texture Type:Default**

Property:	**Function:**
Texture Type	**Default** is the most common setting used for all Textures. It provides access to most of the properties for Texture importing.
Texture Shape	Use this to define the shape of the Texture. See documentation on the Texture Importer for information on all Texture shapes.

Property:	Function:
sRGB (Color Texture)	Check this box to specify that the Texture is stored in gamma space. This should always be checked for non-HDR color Textures (such as Albedo and Specular Color). If the Texture stores information that has a specific meaning, and you need the exact values in the **Shader** (for example, the smoothness or the metalness), uncheck this box. This box is checked by default.
Alpha Source	Use this to specify how the alpha channel of the Texture is generated. This is set to **None** by default.
None	The imported Texture does not have an alpha channel, whether or not the input Texture has one.
Input Texture Alpha	This uses the alpha from the input Texture if a Texture is provided.
From Gray Scale	This generates the alpha from the mean (average) of the input Texture RGB values.
Alpha is Transparency	If the alpha channel you specify is Transparency, enable **Alpha is Transparency** to dilate the color and avoid filtering artifacts on the edges.
Advanced	
Non Power of 2	If the Texture has a non-power of two (NPOT) dimension size, this defines a scaling behavior at import time. See documentation on Importing Textures for more information on non-power of two sizes. This is set to **None** by default.
None	Texture dimension size stays the same.
To nearest	The Texture is scaled to the nearest power-of-two dimension size at import time. For example, a 257x511 px Texture is scaled to 256x512 px. Note that **PVRTC** formats require Textures to be square (that is width equal to height), so the final dimension size is upscaled to 512x512 px.
To larger	The Texture is scaled to the power-of-two dimension size of the largest dimension size value at import time. For example, a 257x511 px Texture is scaled to 512x512 px.
To smaller	The Texture is scaled to the power-of-two dimension size of the smallest dimension size value at import time. For example, a 257x511 px Texture is scaled to 256x256 px.
Read/Write Enabled	Check this box to enable access to the Texture data from script functions (such as Texture2D.SetPixels, Texture2D.GetPixels and other Texture2D functions). Note that a copy of the Texture data is made, doubling the amount of memory required for Texture **Assets**, so only use this property if absolutely necessary. This is only valid for uncompressed and DXT compressed Textures; other types of compressed textures cannot be read from. This property is disabled by default.
Generate Mip Maps	Check this box to enable mipmap generation. Mipmaps are smaller versions of the Texture that get used when the Texture is very small on screen. See documentation on Importing Textures for more information on mipmaps.
Border Mip Maps	Check this box to avoid colors bleeding out to the edge of the lower MIP levels. Used for light cookies (see below). This box is unchecked by default.
Mip Map Filtering	There are two ways of mipmap filtering available for optimizing image quality. The default option is **Box**.
Box	This is the simplest way to fade out mipmaps. The MIP levels become smoother as they go down in dimension size.
Kaiser	A sharpening algorithm runs on the mipmaps as they go down in dimension size. Try this option if your Textures are too blurry in the distance. (The algorothm is of a Kaiser Window type - see Wikipedia for further information.)
Fadeout Mip Maps	Enable this to make the mipmaps fade to gray as the MIP levels progress. This is used for detail maps. The left-most scroll is the first MIP level to begin fading out. The right-most scroll defines the MIP level where the Texture is completely grayed out.
Wrap Mode	Select how the Texture behaves when tiled. The default option is **Clamp**.
Repeat	The Texture repeats itself in tiles.
Clamp	The Texture's edges are stretched.

Property:	Function:
Filter Mode	Select how the Texture is filtered when it gets stretched by 3D transformations. The default option is **Point (no filter)**.
Point (no filter)	The Texture appears blocky up close.
Bilinear	The Texture appears blurry up close.
Trilinear	Like Bilinear, but the Texture also blurs between the different MIP levels.
Aniso Level	Increases Texture quality when viewing the Texture at a steep angle. Good for floor and ground Textures. See documentation on Importing Textures for more information on Anisotropic filtering.

Texture type: Normal Map

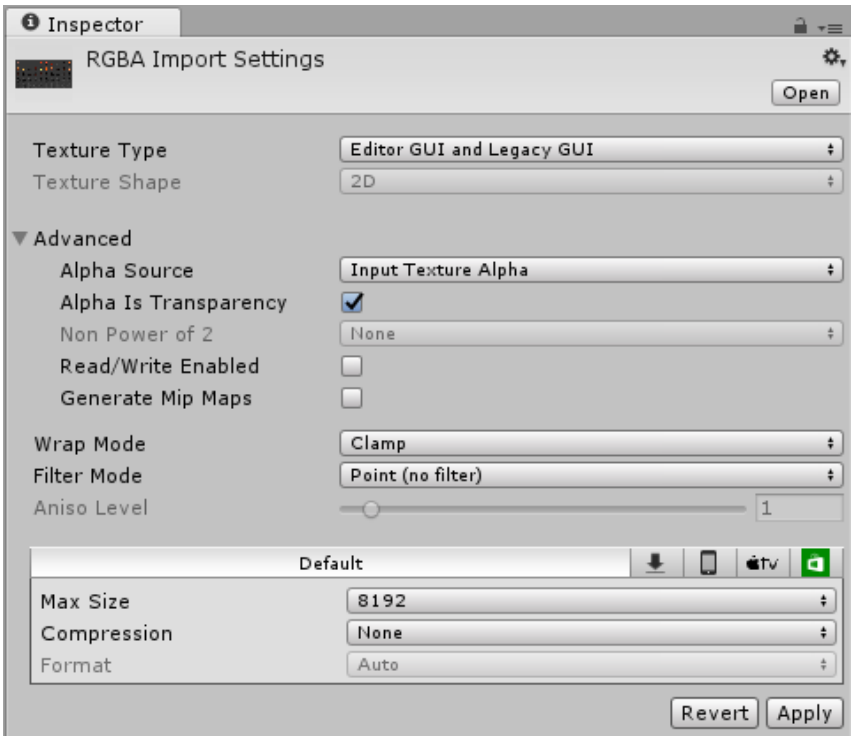


Texture Inspector window - **Texture Type:Normal Map**

Property:	Function:
Texture Type	Select **Normal map** to turn the color channels into a format suitable for real-time normal mapping.
Texture Shape	Use this to define the shape of the Texture. See documentation on the Texture Importer for information on all Texture shapes.
Create from Greyscale	This creates the Normal Map from a greyscale heightmap. Check this to enable it and enabled it and see the **Bumpiness** and **Filtering**. This option is unchecked by default.
Bumpiness	Control the amount of bumpiness. A low bumpiness value means that even sharp contrast in the heightmap is translated to gentle angles and bumps. A high value creates exaggerated bumps and very high-contrast lighting responses to the bumps. This option is only visible if Create from Greyscale is checked.
Filtering	Determine how the bumpiness is calculated:
Smooth	This generates Normal Maps with standard (forward differences) algorithms.

Property:	Function:
Sharp	Also known as a Sobel filter, this generates Normal Maps that are sharper than Standard.
Advanced	
Non Power of 2	If the Texture has a non-power of two (NPOT) dimension size, this defines a scaling behavior at import time. See documentation on Importing Textures for more information on non-power of two dimension sizes. This is set to **None** by default.
None	Texture dimension size stays the same.
To nearest	The Texture is scaled to the nearest power-of-two dimension size at import time. For example, a 257x511 px Texture is scaled to 256x512 px. Note that PVRTC formats require Textures to be square (width equal to height), so the final dimension size is upscaled to 512x512 px.
To larger	The Texture is scaled to the power-of-two dimension size of the largest dimension size value at import time. For example, a 257x511 px Texture is scaled to 512x512 px.
To smaller	The Texture is scaled to the power-of-two dimension size of the smallest dimension size value at import time. For example, a 257x511 px Texture is scaled to 256x256 px.
Read/Write Enabled	Check this box to enable access to the Texture data from script functions (such as Texture2D.SetPixels, Texture2D.GetPixels and other Texture2D functions). Note that a copy of the Texture data is made, doubling the amount of memory required for Texture Assets, so only use this property if absolutely necessary. This is only valid for uncompressed and DXT compressed Textures; other types of compressed textures cannot be read from. This property is disabled by default.
Generate Mip Maps	Check this box to enable mipmap generation. Mipmaps are smaller versions of the Texture that get used when the Texture is very small on screen. See documentation on Importing Textures for more information on mipmaps.
Border Mip Maps	Check this box to avoid colors bleeding out to the edge of the lower MIP levels. Used for light cookies (see below). This box is unchecked by default.
Mip Map Filtering	There are two ways of mipmap filtering available for optimizing image quality. The default option is **Box**.
Box	This is the simplest way to fade out mipmaps. The MIP levels become smoother as they go down in dimension size.
Kaiser	A sharpening algorithm runs on the mipmaps as they go down in dimension size. Try this option if your Textures are too blurry in the distance. (The algorothm is of a Kaiser Window type - see Wikipedia for further information.)
Fadeout Mip Maps	Enable this to make the mipmaps fade to gray as the MIP levels progress. This is used for detail maps. The left-most scroll is the first MIP level to begin fading out. The right-most scroll defines the MIP level where the Texture is completely grayed out.
Wrap Mode	Select how the Texture behaves when tiled. The default option is **Clamp**.
Repeat	The Texture repeats itself in tiles.
Clamp	The Texture's edges are stretched.
Filter Mode	Select how the Texture is filtered when it gets stretched by 3D transformations. The default option is **Point (no filter)**.
Point (no filter)	The Texture appears blocky up close.
Bilinear	The Texture appears blurry up close.
Trilinear	Like Bilinear, but the Texture also blurs between the different MIP levels.
Aniso Level	Increases Texture quality when viewing the Texture at a steep angle. Good for floor and ground Textures. See documentation on Importing Textures for more information on Anisotropic filtering.

Texture type: Editor GUI and Legacy GUI

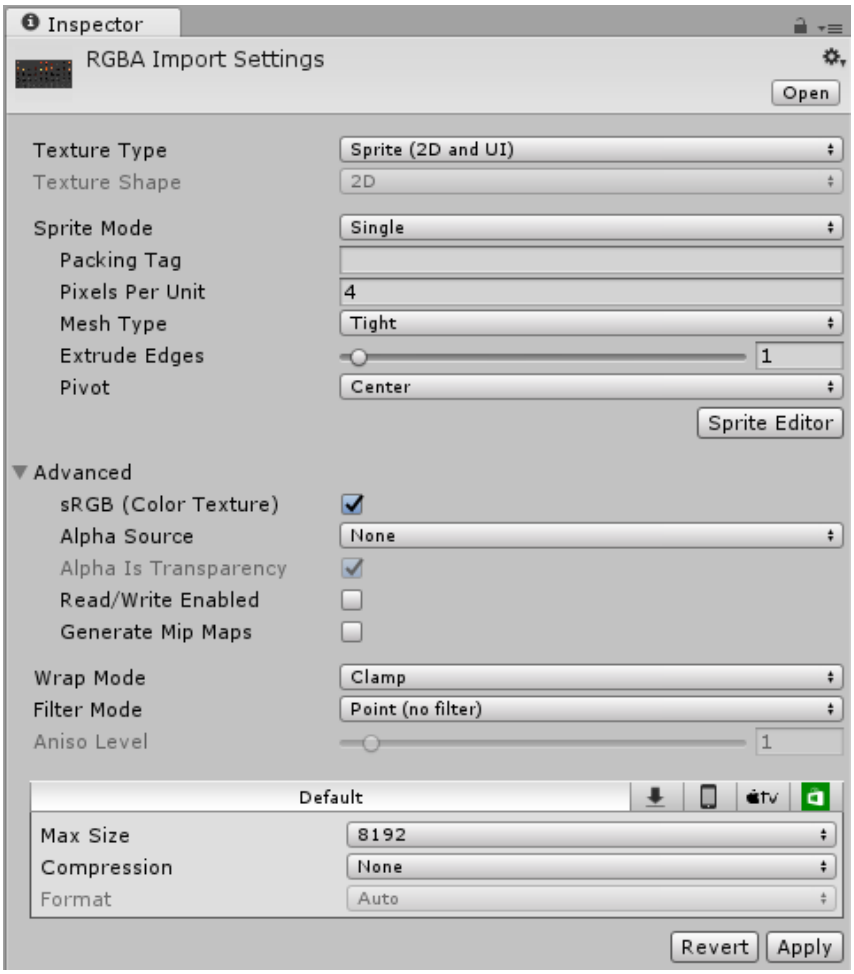


Texture Inspector window - **Texture Type:Editor GUI and Legacy GUI**

Property:	Function:
Texture Type	Select **Editor GUI and Legacy GUI** if you are using the Texture on any HUD or GUI controls.
Texture Shape	Use this to define the shape of the Texture. See documentation on the Texture Importer for information on all Texture shapes.
Advanced	
Alpha Source	Use this to specify how the alpha channel of the Texture is generated. This is set to None by default.
None	The imported Texture does not have an alpha channel, whether or not the input Texture has one.
Input Texture Alpha	This uses the alpha from the input Texture if a Texture is provided.
From Gray Scale	This generates the alpha from the mean (average) of the input Texture RGB values.
Alpha is Transparency	If the alpha channel you specify is Transparency, enable **Alpha is Transparency** to dilate the color and avoid filtering artifacts on the edges.
Non Power of 2	If the Texture has a non-power of two (NPOT) dimension size, this defines a scaling behavior at import time. See documentation on Importing Textures for more information on non-power of two sizes. This is set to **None** by default.
None	Texture dimension size stays the same.
To nearest	The Texture is scaled to the nearest power-of-two dimension size at import time. For example, a 257x511 px Texture is scaled to 256x512 px. Note that PVRTC formats require Textures to be square (width equal to height), so the final dimension size is upscaled to 512x512 px.
To larger	The Texture is scaled to the power-of-two dimension size of the largest dimension size value at import time. For example, a 257x511 px Texture is scaled to 512x512 px.
To smaller	The Texture is scaled to the power-of-two dimension size of the smallest dimension size value at import time. For example, a 257x511px Texture is scaled to 256x256.

Property:	Function:
Read/Write Enabled	Check this box to enable access to the Texture data from script functions (such as Texture2D.SetPixels, Texture2D.GetPixels and other Texture2D functions). Note that a copy of the Texture data is made, doubling the amount of memory required for Texture Assets, so only use this property if absolutely necessary. This is only valid for uncompressed and DXT compressed Textures; other types of compressed textures cannot be read from. This property is disabled by default.
Generate Mip Maps	Check this box to enable mipmap generation. Mipmaps are smaller versions of the Texture that get used when the Texture is very small on screen. See documentation on Importing Textures for more information on mipmaps.
Border Mip Maps	Check this box to avoid colors bleeding out to the edge of the lower MIP levels. Used for light cookies (see below). This box is unchecked by default.
Mip Map Filtering	There are two ways of mipmap filtering available for optimizing image quality. The default option is **Box**.
Box	This is the simplest way to fade out mipmaps. The MIP levels become smoother as they go down in dimension size.
Kaiser	A sharpening algorithm runs on the mipmaps as they go down in dimension size. Try this option if your Textures are too blurry in the distance. (The algorothm is of a Kaiser Window type - see Wikipedia for further information.)
Fadeout Mip Maps	Enable this to make the mipmaps fade to gray as the MIP levels progress. This is used for detail maps. The left-most scroll is the first MIP level to begin fading out. The right-most scroll defines the MIP level where the Texture is completely grayed out.
Wrap Mode	Select how the Texture behaves when tiled. The default option is **Clamp**.
Repeat	The Texture repeats itself in tiles.
Clamp	The Texture's edges are stretched.
Filter Mode	Select how the Texture is filtered when it gets stretched by 3D transformations. The default option is **Point (no filter)**.
Point (no filter)	The Texture appears blocky up close.
Bilinear	The Texture appears blurry up close.
Trilinear	Like Bilinear, but the Texture also blurs between the different MIP levels.
Aniso Level	Increases Texture quality when viewing the Texture at a steep angle. Good for floor and ground Textures. See documentation on Importing Textures for more information on Anisotropic filtering.

Texture type: Sprite (2D and UI)

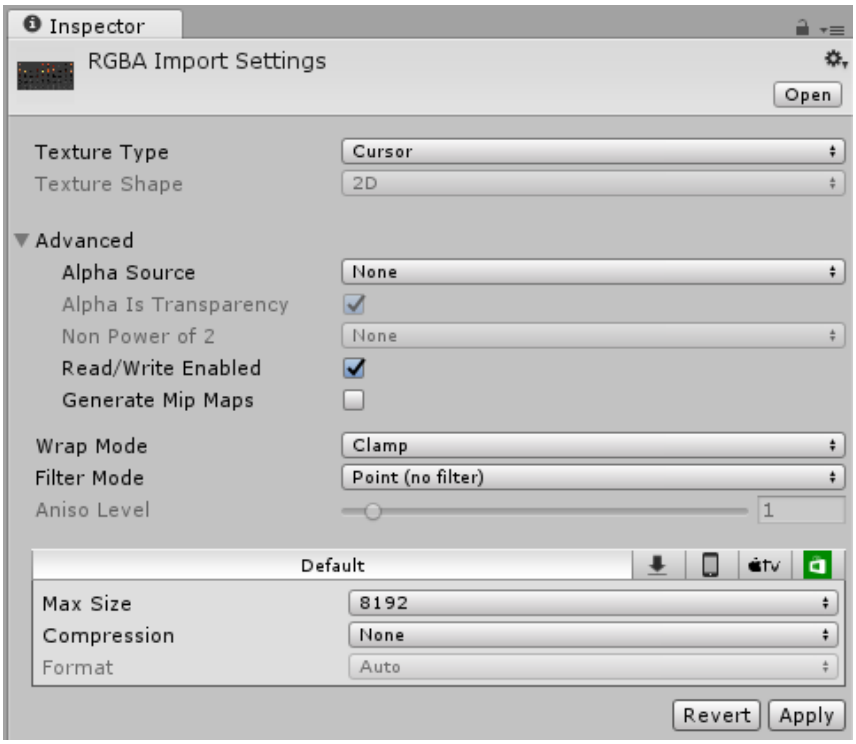


Texture Inspector window - **Texture Type:Sprite (2D and UI)**

Property:	Function:
Texture Type	Select **Sprite (2D and UI)** if you are using the Texture in a 2D game as a Sprite.
Texture Shape	Use this to define the shape of the Texture. See documentation on the Texture Importer for information on all Texture shapes.
Sprite mode	Use this setting to specify how the the Sprite graphic is extracted from the image. The default for this option is **Single**.
Single	Use the Sprite image in isolation.
Multiple	Keep multiple related Sprites together in the same image (for example, animation frames or separate Sprite elements that belong to a single game character).
Packing Tag	Specify by name a Sprite atlas which you want to pack this Texture into.
Pixels Per Unit	The number of **pixels** of width/height in the Sprite image that correspond to one distance unit in world space.
Mesh Type	This defines the **Mesh** type that is generated for the Sprite. The default for this option is **Tight**.
Full Rect	This creates a **quad** to map the Sprite onto it.
Tight	This generates a Mesh based on pixel alpha value. The Mesh generated generally follows the shape of the Sprite. **Note:** Any Sprite that is smaller than 32x32 uses **Full Rect**, even when **Tight** is specified.
Extrude Edges	Use the slider to determine how much area to leave around the Sprite in the generated Mesh.
Pivot	The location in the image where the Sprite's local coordinate system originates. Choose one of the pre-set options, or select **Custom** to set your own Pivot location.
Custom	Define the X and Y to set a custom Pivot location in the image.
Advanced	

Property:	Function:
sRGB (Color Texture)	Use this to specify whether or not the Texture is stored in gamma space. This should be the case for all non-HDR color Textures (such as Albedo and Specular Color). If the Texture stores information that has a specific meaning, and you need the exact values in the Shader (for example, the smoothness or the metalness), uncheck this box.
Alpha Source	Use this to specify how the alpha channel of the Texture is generated.
None	The imported Texture does not have an alpha channel, whether or not the input Texture has one.
Input Texture Alpha	This uses the alpha from the input Texture. This option does not appear in the menu if there is no alpha in the imported Texture.
From Gray Scale	This generates the alpha from the mean (average) of the input Texture RGB values.
Alpha is Transparency	If the provided alpha channel is Transparency, enable Alpha is Transparency to dilate the color and avoid filtering artifacts on the edges.
Read/Write Enabled	Check this box to enable access to the Texture data from script functions (such as Texture2D.SetPixels, Texture2D.GetPixels and other Texture2D functions). Note however that a copy of the Texture data will be made, doubling the amount of memory required for Texture Asset, so only use this property if absolutely necessary. This is only valid for uncompressed and DXT compressed Textures; other types of compressed textures cannot be read from. This property is disabled by default.
Generate Mip Maps	Check this box to enable mipmap generation. Mipmaps are smaller versions of the Texture that get used when the Texture is very small on screen. See the Details section at the end of the page.
Border Mip Maps	Select this to avoid colors bleeding out to the edge of the lower MIP levels. Used for Light Cookies (see below).
Mip Map Filtering	There are two ways of mipmap filtering available for optimizing image quality:
Box	This is the simplest way to fade out mipmaps. The MIP levels become smoother as they go down in dimension size.
Kaiser	A sharpening algorithm runs on the mipmaps as they go down in dimension size. Try this option if your Textures are too blurry in the distance. (The algorothm is of a Kaiser Window type - see Wikipedia for further information.)
Fadeout Mip Maps	Enable this to make the mipmaps fade to gray as the MIP levels progress. This is used for detail maps. The left-most scroll is the first MIP level to begin fading out. The right-most scroll defines the MIP level where the Texture is completely grayed out.
Wrap Mode	Select how the Texture behaves when tiled:
Repeat	The Texture repeats itself in tiles.
Clamp	The Texture's edges are stretched.
Filter Mode	Select how the Texture is filtered when it gets stretched by 3D transformations:
Point	The Texture appears blocky up close.
Bilinear	The Texture appears blurry up close.
Trilinear	Like Bilinear, but the Texture also blurs between the different MIP levels.
Aniso Level	Increases Texture quality when viewing the Texture at a steep angle. Good for floor and ground Textures. See the Details section at the end of the page.

Texture type: Cursor

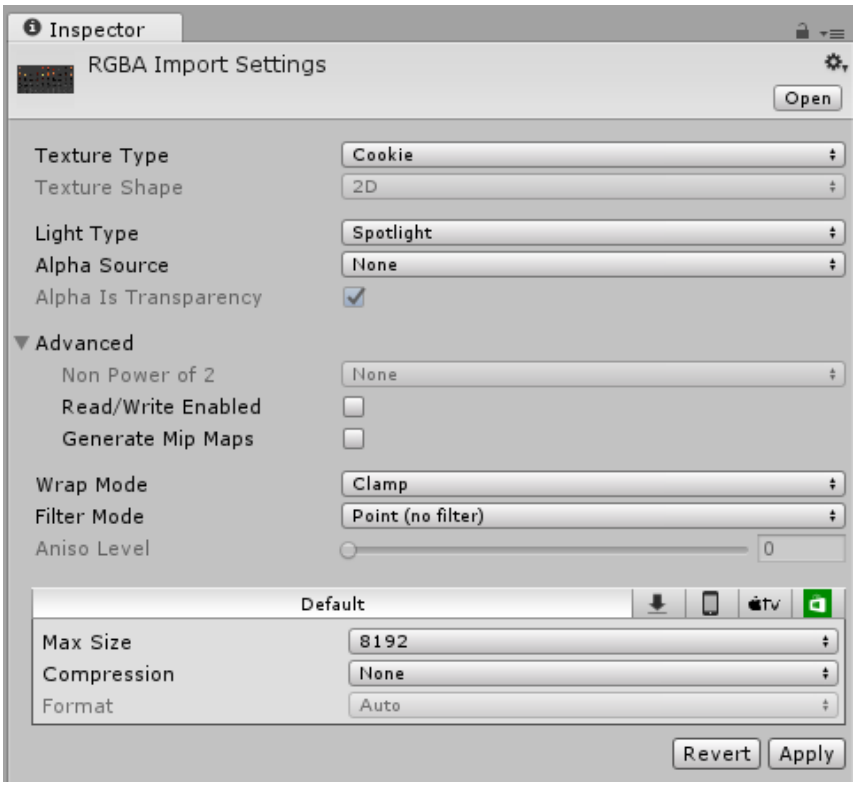


Texture Inspector window - **Texture Type:Cursor**

Property:	Function:
Texture Type	Select **Cursor** if you are using the Texture as a custom cursor.
Texture Shape	Use this to define the shape of the Texture. See documentation on the Texture Importer for information on all Texture shapes.
Advanced	
Alpha Source	Use this to specify how the alpha channel of the Texture is generated. This is set to None by default.
None	The imported Texture does not have an alpha channel, whether or not the input Texture has one.
Input Texture Alpha	This uses the alpha from the input Texture if a Texture is provided.
From Gray Scale	This generates the alpha from the mean (average) of the input Texture RGB values.
Alpha is Transparency	If the alpha channel you specify is Transparency, enable **Alpha is Transparency** to dilate the color and avoid filtering artifacts on the edges.
Non Power of 2	If the Texture has a non-power of two (NPOT) dimension size, this defines a scaling behavior at import time. See documentation on Importing Textures for more information on non-power of two dimension sizes. This is set to **None** by default.
None	Texture dimension size stays the same.
To nearest	The Texture is scaled to the nearest power-of-two dimension size at import time. For example, a 257x511 px Texture is scaled to 256x512 px. Note that PVRTC formats require Textures to be square (width equal to height), so the final dimension size is upscaled to 512x512 px.
To larger	The Texture is scaled to the power-of-two dimension size of the largest dimension size value at import time. For example, a 257x511 px Texture is scaled to 512x512 px.
To smaller	The Texture is scaled to the power-of-two dimension size of the smallest dimension size value at import time. For example, a 257x511 px Texture is scaled to 256x256.

Property:	Function:
Read/Write Enabled	Check this box to enable access to the Texture data from script functions (such as Texture2D.SetPixels, Texture2D.GetPixels and other Texture2D functions). Note that a copy of the Texture data is made, doubling the amount of memory required for Texture Assets, so only use this property if absolutely necessary. This is only valid for uncompressed and DXT compressed Textures; other types of compressed textures cannot be read from. This property is disabled by default.
Generate Mip Maps	Check this box to enable mipmap generation. Mipmaps are smaller versions of the Texture that get used when the Texture is very small on screen. See documentation on Importing Textures for more information on mipmaps.
Border Mip Maps	Check this box to avoid colors bleeding out to the edge of the lower MIP levels. Used for light cookies (see below). This box is unchecked by default.
Mip Map Filtering	There are two ways of mipmap filtering available for optimizing image quality. The default option is **Box**.
Box	This is the simplest way to fade out mipmaps. The MIP levels become smoother as they go down in dimension size.
Kaiser	A sharpening algorithm runs on the mipmaps as they go down in dimension size. Try this option if your Textures are too blurry in the distance. (The algorothm is of a Kaiser Window type - see Wikipedia for further information.)
Fadeout Mip Maps	Enable this to make the mipmaps fade to gray as the MIP levels progress. This is used for detail maps. The left-most scroll is the first MIP level to begin fading out. The right-most scroll defines the MIP level where the Texture is completely grayed out.
Wrap Mode	Select how the Texture behaves when tiled. The default option is **Clamp**.
Repeat	The Texture repeats itself in tiles.
Clamp	The Texture's edges are stretched.
Filter Mode	Select how the Texture is filtered when it gets stretched by 3D transformations. The default option is **Point (no filter)**.
Point (no filter)	The Texture appears blocky up close.
Bilinear	The Texture appears blurry up close.
Trilinear	Like Bilinear, but the Texture also blurs between the different MIP levels.
Aniso Level	Increases Texture quality when viewing the Texture at a steep angle. Good for floor and ground Textures. See documentation on Importing Textures for more information on Anisotropic filtering.

Texture type: Cookie

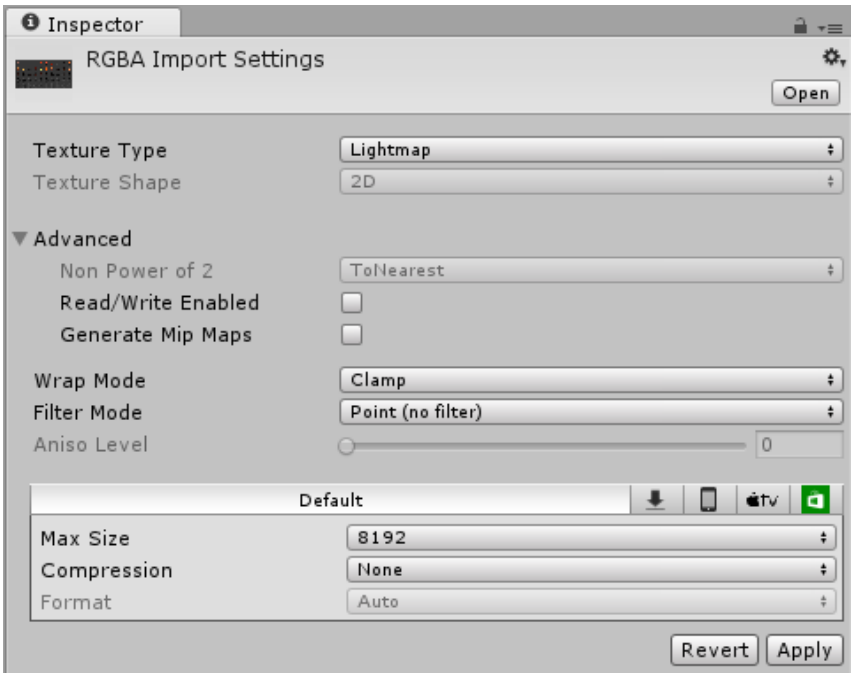


Texture Inspector window - **Texture Type:Cookie**

Property:	Function:
Texture Type	Select **Cookie** to set your Texture up with the basic parameters used for the Cookies of your **Scene**'s Lights.
Texture Shape	Use this to define the shape of the Texture. See documentation on the Texture Importer for information on all Texture shapes.
Light Type	Define the type of Light that the Texture is applied to. Directional and Spotlight cookies must be 2D Textures, and Point Light Cookies must be cubemaps. The system automatically enforces the right shape depending on the Light type. For Directional Lights this Texture tiles, so in the Texture inspector set the **Edge Mode** to **Repeat**. For Spotlights, keep the edges of your Cookie Texture solid black to get the proper effect. In the Texture Inspector, set the **Edge Mode** to **Clamp**.
Alpha is Transparency	If the alpha channel you specify is Transparency, enable **Alpha is Transparency** to dilate the color and avoid filtering artifacts on the edges.
Advanced	
Non Power of 2	If the Texture has a non-power of two (NPOT) dimension size, this defines a scaling behavior at import time. See documentation on Importing Textures for more information on non-power of two sizes. This is set to **None** by default.
None	Texture dimension size stays the same.
To nearest	The Texture is scaled to the nearest power-of-two dimension size at import time. For example, a 257x511 px Texture is scaled to 256x512 px. Note that PVRTC formats require Textures to be square (width equal to height), so the final dimension size is upscaled to 512x512 px.
To larger	The Texture is scaled to the power-of-two dimension size of the largest dimension size value at import time. For example, a 257x511 px Texture is scaled to 512x512 px.
To smaller	The Texture is scaled to the power-of-two dimension size of the smallest dimension size value at import time. For example, a 257x511 px Texture is scaled to 256x256.

Property:	Function:
Read/Write Enabled	Check this box to enable access to the Texture data from script functions (such as Texture2D.SetPixels, Texture2D.GetPixels and other Texture2D functions). Note that a copy of the Texture data is made, doubling the amount of memory required for Texture Assets, so only use this property if absolutely necessary. This is only valid for uncompressed and DXT compressed Textures; other types of compressed textures cannot be read from. This property is disabled by default.
Generate Mip Maps	Check this box to enable mipmap generation. Mipmaps are smaller versions of the Texture that get used when the Texture is very small on screen. See documentation on Importing Textures for more information on mipmaps.
Border Mip Maps	Check this box to avoid colors bleeding out to the edge of the lower MIP levels. Used for light cookies (see below). This box is unchecked by default.
Mip Map Filtering	There are two ways of mipmap filtering available for optimizing image quality. The default option is **Box**.
Box	This is the simplest way to fade out mipmaps. The MIP levels become smoother as they go down in dimension size.
Kaiser	A sharpening algorithm runs on the mipmaps as they go down in dimension size. Try this option if your Textures are too blurry in the distance. (The algorothm is of a Kaiser Window type - see Wikipedia for further information.)
Fadeout Mip Maps	Enable this to make the mipmaps fade to gray as the MIP levels progress. This is used for detail maps. The left-most scroll is the first MIP level to begin fading out. The right-most scroll defines the MIP level where the Texture is completely grayed out.
Wrap Mode	Select how the Texture behaves when tiled. The default option is **Clamp**.
Repeat	The Texture repeats itself in tiles.
Clamp	The Texture's edges are stretched.
Filter Mode	Select how the Texture is filtered when it gets stretched by 3D transformations. The default option is **Point (no filter)**.
Point (no filter)	The Texture appears blocky up close.
Bilinear	The Texture appears blurry up close.
Trilinear	Like Bilinear, but the Texture also blurs between the different MIP levels.
Aniso Level	Increases Texture quality when viewing the Texture at a steep angle. Good for floor and ground Textures. See documentation on Importing Textures for more information on Anisotropic filtering.

Texture type: Lightmap

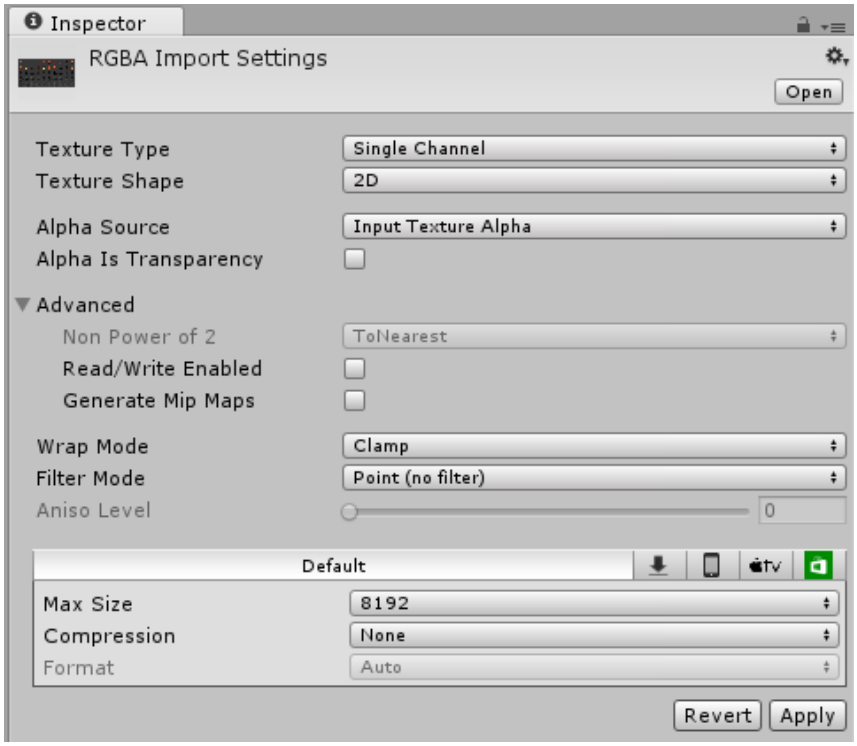


Texture Inspector window - **Texture Type:Lightmap**

Property:	Function:
Texture Type	Select **Lightmap** if you are using the Texture as a Lightmap. This option enables encoding into a specific format (RGBM or dLDR, depending on the platform) and a post-processing step on Texture data (a push-pull dilation pass).
Texture Shape	Use this to define the shape of the Texture. See documentation on the Texture Importer for information on all Texture shapes.
Advanced	
Non Power of 2	If the Texture has a non-power of two (NPOT) dimension size, this defines a scaling behavior at import time. See documentation on Importing Textures for more information on non-power of two dimension sizes. This is set to **None** by default.
None	Texture size stays the same.
To nearest	The Texture is scaled to the nearest power-of-two dimension size at import time. For example, a 257x511 px Texture is scaled to 256x512 px. Note that PVRTC formats require Textures to be square (width equal to height), so the final dimension size is upscaled to 512x512 px.
To larger	The Texture is scaled to the power-of-two dimension size of the largest size value at import time. For example, a 257x511 px Texture is scaled to 512x512 px.
To smaller	The Texture is scaled to the power-of-two dimension size of the smallest dimension size value at import time. For example, a 257x511 px Texture is scaled to 256x256 px.
Read/Write Enabled	Check this box to enable access to the Texture data from script functions (such as Texture2D.SetPixels, Texture2D.GetPixels and other Texture2D functions). Note that a copy of the Texture data is made, doubling the amount of memory required for Texture Assets, so only use this property if absolutely necessary. This is only valid for uncompressed and DXT compressed Textures; other types of compressed textures cannot be read from. This property is disabled by default.
Generate Mip Maps	Check this box to enable mipmap generation. Mipmaps are smaller versions of the Texture that get used when the Texture is very small on screen. See documentation on Importing Textures for more information on mipmaps.
Border Mip Maps	Check this box to avoid colors bleeding out to the edge of the lower MIP levels. Used for light cookies (see below). This box is unchecked by default.
Mip Map Filtering	There are two ways of mipmap filtering available for optimizing image quality. The default option is **Box**.
Box	This is the simplest way to fade out mipmaps. The MIP levels become smoother as they go down in dimension size.

Property:	Function:
Kaiser	A sharpening algorithm runs on the mipmaps as they go down in dimension size. Try this option if your Textures are too blurry in the distance. (The algorothm is of a Kaiser Window type - see Wikipedia for further information.)
Fadeout Mip Maps	Enable this to make the mipmaps fade to gray as the MIP levels progress. This is used for detail maps. The left-most scroll is the first MIP level to begin fading out. The right-most scroll defines the MIP level where the Texture is completely grayed out.
Wrap Mode	Select how the Texture behaves when tiled. The default option is **Clamp**.
Repeat	The Texture repeats itself in tiles.
Clamp	The Texture's edges are stretched.
Filter Mode	Select how the Texture is filtered when it gets stretched by 3D transformations. The default option is **Point (no filter)**.
Point (no filter)	The Texture appears blocky up close.
Bilinear	The Texture appears blurry up close.
Trilinear	Like Bilinear, but the Texture also blurs between the different MIP levels.
Aniso Level	Increases Texture quality when viewing the Texture at a steep angle. Good for floor and ground Textures. See documentation on Importing Textures for more information on Anisotropic filtering.

Texture type: Single Channel



Texture Inspector window - **Texture Type:Single Channel**

Property:	Function:
Texture Type	Select **Single Channel** if you only need one channel in the Texture.
Texture Shape	Use this to define the shape of the Texture. See documentation on the Texture Importer for information on all Texture shapes.
Alpha Source	Use this to specify how the alpha channel of the Texture is generated. This is set to None by default.
None	The imported Texture does not have an alpha channel, whether or not the input Texture has one.

Property:	Function:
Input Texture Alpha	This uses the alpha from the input Texture if a Texture is provided.
From Gray Scale	This generates the alpha from the mean (average) of the input Texture RGB values.
Alpha is Transparency	If the alpha channel you specify is Transparency, enable **Alpha is Transparency** to dilate the color and avoid filtering artifacts on the edges.
Advanced	
Non Power of 2	If the Texture has a non-power of two (NPOT) dimension size, this defines a scaling behavior at import time. See documentation on Importing Textures for more information on non-power of two sizes. This is set to **None** by default.
None	Texture dimension size stays the same.
To nearest	The Texture is scaled to the nearest power-of-two dimension size at import time. For example, a 257x511 px Texture is scaled to 256x512 px. Note that PVRTC formats require Textures to be square (width equal to height), so the final dimension size is upscaled to 512x512 px.
To larger	The Texture is scaled to the power-of-two dimension size of the largest dimension size value at import time. For example, a 257x511 px Texture is scaled to 512x512 px.
To smaller	The Texture is scaled to the power-of-two dimension size of the smallest size value at import time. For example, a 257x511 px Texture is scaled to 256x256.
Read/Write Enabled	Check this box to enable access to the Texture data from script functions (such as Texture2D.SetPixels, Texture2D.GetPixels and other Texture2D functions). Note that a copy of the Texture data is made, doubling the amount of memory required for Texture Assets, so only use this property if absolutely necessary. This is only valid for uncompressed and DXT compressed Textures; other types of compressed textures cannot be read from. This property is disabled by default.
Generate Mip Maps	Check this box to enable mipmap generation. Mipmaps are smaller versions of the Texture that get used when the Texture is very small on screen. See documentation on Importing Textures for more information on mipmaps.
Border Mip Maps	Check this box to avoid colors bleeding out to the edge of the lower MIP levels. Used for light cookies (see below). This box is unchecked by default.
Mip Map Filtering	There are two ways of mipmap filtering available for optimizing image quality. The default option is **Box**.
Box	This is the simplest way to fade out mipmaps. The MIP levels become smoother as they go down in dimension size.
Kaiser	A sharpening algorithm runs on the mipmaps as they go down in dimension size. Try this option if your Textures are too blurry in the distance. (The algorothm is of a Kaiser Window type - see Wikipedia for further information.)
Fadeout Mip Maps	Enable this to make the mipmaps fade to gray as the MIP levels progress. This is used for detail maps. The left-most scroll is the first MIP level to begin fading out. The right-most scroll defines the MIP level where the Texture is completely grayed out.
Wrap Mode	Select how the Texture behaves when tiled. The default option is **Clamp**.
Repeat	The Texture repeats itself in tiles.
Clamp	The Texture's edges are stretched.
Filter Mode	Select how the Texture is filtered when it gets stretched by 3D transformations. The default option is **Point (no filter)**.
Point (no filter)	The Texture appears blocky up close.
Bilinear	The Texture appears blurry up close.
Trilinear	Like Bilinear, but the Texture also blurs between the different MIP levels.

Property:	Function:
Aniso Level	Increases Texture quality when viewing the Texture at a steep angle. Good for floor and ground Textures. See documentation on Importing Textures for more information on Anisotropic filtering.

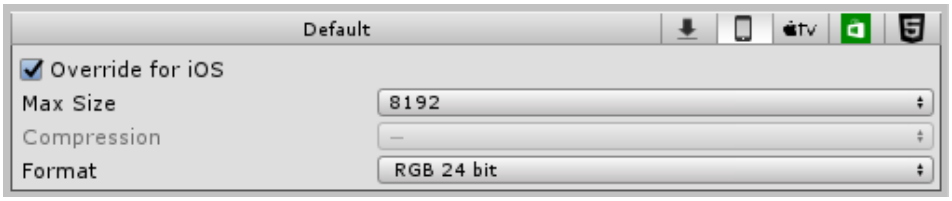
Texture compression formats for platform-specific overrides

Leave feedback

While Unity supports many common image formats as source files for importing your Textures (such as JPG, PNG, PSD and TGA), these formats are not used during realtime **rendering** by 3D graphics hardware such as a graphics card or mobile device. 3D graphics hardware requires Textures to be compressed in specialized formats which are optimised for fast Texture sampling. The various different platforms and devices available each have their own different proprietary formats.

By default, the Unity Editor automatically converts Textures to the most appropriate format to match the build target you have selected. Only the converted Textures are included in your build; your source **Asset** files are left in their original format, in your project's *Assets* folder. However, on most platforms there are a number of different supported Texture **compression** formats to choose from. Unity has certain default formats set up for each platform, but in some situations you may want to override the default and pick a different **compression** format for some of your Textures (for example, if you are using a Texture as a mask, with only one channel, you might choose to use the BC4 format to save space while preserving quality).

To apply custom settings for each platform, use the Texture Importer to set default options, then use the **Platform-specific overrides** panel to override those defaults for specific platforms.



Default internal Texture representation per platform

The following table shows the default formats used for each platform.

Platform	Color model	None	Normal quality (Default)	High quality	Low quality (higher performance)
Windows, Linux, macOS, **PS4**, **XBox One**	RGB	RGB 24 bit	RGB Compressed DXT1	RGB(A) Compressed BC7	RGB Compressed DXT1
	RGBA	RGBA 32 bit	RGBA Compressed DXT5	RGB(A) Compressed BC7	RGBA Compressed DXT5
	HDR	RGBA Half	RGB Compressed BC6H	RGB Compressed BC6H	RGB Compressed BC6H
WebGL	RGB	RGB 24 bit	RGB Compressed DXT1	RGB Compressed DXT1	RGB Compressed DXT1
	RGBA	RGBA 32 bit	RGBA Compressed DXT5	RGBA Compressed DXT5	RGBA Compressed DXT5
Android (default subTarget)	RGB	RGB 24 bit	RGB Compressed **ETC**	RGB Compressed **ETC**	RGB Compressed **ETC**
	RGBA	RGBA 32 bit	RGBA Compressed ETC2	RGBA Compressed ETC2	RGBA Compressed ETC2
iOS	RGB	RGB 24 bit	RGB Compressed **PVRTC** 4 bits	RGB Compressed **PVRTC** 4 bits	RGB Compressed **PVRTC** 2 bits
	RGBA	RGBA 32 bit	RGBA Compressed PVRTC 4 bits	RGBA Compressed PVRTC 4 bits	RGBA Compressed PVRTC 2 bits
tvOS	RGB	RGB 24 bit	RGB Compressed **ASTC** 6x6 block	RGB Compressed **ASTC** 4x4 block	RGB Compressed **ASTC** 8x8 block

Platform	Color model	None	Normal quality (Default)	High quality	Low quality (higher performance)
	RGBA	RGBA 32 bit	RGBA Compressed ASTC 6x6 block	RGBA Compressed ASTC 4x4 block	RGBA Compressed ASTC 8x8 block
	RGBA	RGBA 32 bit	RGBA 16 bit	RGBA 16 bit	RGBA 16 bit

All supported Texture compression formats

The following table shows the **Texture compression** format options available on each platform, and the resulting compressed file size (based on a 256px-square image). Choosing a Texture compression format is a balance between file size and quality; the higher the quality, the greater the file size. In the description below, see the final file size of a in-game Texture of 256 by 256 **pixels**.

When you use a Texture compression format that is not supported on the target platform, the Textures are decompressed to RGBA 32 and stored in memory alongside the compressed Textures. When this happens, time is lost decompressing Textures, and memory is lost because you are storing them twice. In addition, all platforms have different hardware, and are optimised to work most efficiently with specific compression formats; choosing non-compatible formats can impact your game's performance. The table below shows supported platforms for each compression format.

Notes on the table below:

RGB is a color model in which red, green and blue are added together in various ways to reproduce a broad array of colors.

RGBA is a version of **RGB** with an alpha channel, which supports blending and opacity alteration.

Crunch compression is a lossy compression format (meaning that parts of the data are lost during compression) on top of DXT or ETC Texture compression. Textures are decompressed to DXT or ETC on the CPU, and then uploaded to the GPU at run time. Crunch compression helps the Texture use the lowest possible amount of space on disk and for downloads. Crunch Textures can take a long time to compress, but decompression at run time is very fast.

Texture compression format	Description	Size for a 256x256 pixel Texture	Platform Support
RGB Compressed DXT1	Compressed unsigned normalised integer RGB Texture.	32KB (4 bits per pixel)	Windows, Linux, macOS, PS4, XBox One, Android (Nvidia Tegra and Intel Bay Trail), WebGL Note: With linear rendering on web browser that doesn't support sRGB DXT, textures are uncompressed at run time to RGBA32.
RGB Crunched DXT1	Similar to RGB Compressed DXT1, but compressed using Crunch compression. See Notes, above, for more on crunch compression.	Variable, depending on the complexity of the content in the texture.	Windows, Linux, macOS, PS4, XBox One, Android (Nvidia Tegra and Intel Bay Trail), WebGL Note: With linear rendering on a web browser that doesn't support sRGB DXT, textures are uncompressed at run time to RGBA32.

Texture compression format	Description	Size for a 256x256 pixel Texture	Platform Support
RGBA Compressed DXT5	Compressed unsigned normalised integer RGBA Texture. 8 bits per pixel.	64KB (8 bits per pixel)	Windows, Linux, macOS, PS4, XBox One, Android (Nvidia Tegra and Intel Bay Trail), WebGL Note: With linear rendering on a web browser that doesn't support sRGB DXT, textures are uncompressed at run time to RGBA32.
RGBA Crunched DXT5	Similar to RGBA Compressed DXT5, but compressed using Crunch compression. See Notes, above, for more on crunch compression.	Variable, depending on the complexity of the content in the texture.	Windows, Linux, macOS, PS4, XBox One, Android (Nvidia Tegra and Intel Bay Trail), WebGL Note: With linear rendering on a web browser that doesn't support sRGB DXT, textures are uncompressed at run time to RGBA32.
RGB Compressed BC6H	Compressed unsigned float/High Dynamic Range (HDR) RGB Texture.	64KB (8 bits per pixel)	Windows Direct3D 11: OpenGL 4, Linux. Note: BC6H Textures are uncompressed at run time to RGBA half on the following platform configurations: - macOS with OpenGL - Platforms with Direct3D 10 Shader Model 4 or OpenGL 3 GPUs. Windows Direct3D 11: OpenGL 4, Linux
RGB(A) Compressed BC7	High-quality compressed unsigned normalised integer RGB or RGBA Texture.	64KB (8 bits per pixel)	Note: BC7 Textures are uncompressed at run time to RGBA 32bits on the following platform configurations: - macOS with OpenGL - Platforms with Direct3D 10 Shader Model 4 Platforms with OpenGL 3 GPUs.
RGB Compressed ETC	Compressed RGB Texture. This is the default texture compression format for textures without an alpha channel for Android projects.	32KB (4 bits per pixel)	Android, iOS, tvOS. Note: ETC1 is supported by all OpenGL ES 2.0 GPUs. It does not support alpha.
RGB Crunched ETC	Similar to RGB Compressed ETC, but compressed using Crunch compression. See Notes above for more on crunch compression.	Variable, depending on the complexity of the content in the texture.	Android, iOS, tvOS.

Texture compression format	Description	Size for a 256x256 pixel Texture	Platform Support
RGB Compressed ETC2	Compressed RGB Texture.	32KB (4 bits per pixel)	Android (OpenGL ES 3.0) Note: On Android platforms that don't support ETC2, the texture is uncompressed at run time to the format specified by **ETC2 fallback** in the Build Settings.
RGBA Compressed ETC2	Compressed RGBA Texture. This is the default Texture compression format for textures with alpha channel for Android projects.	64KB (8 bits per pixel)	Android (OpenGL ES 3.0), iOS (OpenGL ES 3.0), tvOS (OpenGL ES 3.0) Note: On iOS and tvOS devices that don't support ETC2, the texture is uncompressed at run time to RGBA32. On Android platforms that don't support ETC2, the texture is uncompressed at run time to the format specified by **ETC2 fallback** in the Build Settings.
RGBA Crunched ETC2	Similar to RGBA Compressed ETC2, but compressed using Crunch compression. See Notes above for more on crunch compression.	Variable, depending on the complexity of the content in the texture.	Android (OpenGL ES 3.0), iOS (OpenGL ES 3.0), tvOS (OpenGL ES 3.0) Note: On iOS and tvOS devices that don't support ETC2, the texture is uncompressed at run time to RGBA32. On Android platforms that don't support ETC2, the texture is uncompressed at run time to the format specified by **ETC2 fallback** in the Build Settings.

Texture compression format	Description	Size for a 256x256 pixel Texture	Platform Support
RGB Compressed ASTC	Variable block size compressed RGB Texture.	12x12: 0.89 bits per pixel (7.56KB for a 256x256 Texture) 10x10: 1.28 bits per pixel (10.56KB for a 256x256 Textures) 8x8: 2 bits per pixel (16KB for a 256x256 Texture); 6x6: 3.56 bits per pixel (28.89KB for a 256x256 Texture) 5x5: 5.12 bits per pixel (42.25KB for a 256x256 Texture) 4x4: 8 bits per pixel (64KB for a 256x256 Texture)	tvOS (all), iOS (A8), Android (PowerVR 6XT, Mali T600 series, Adreno 400 series, Tegra K1).

Texture compression format	Description	Size for a 256x256 pixel Texture	Platform Support
RGBA Compressed ASTC	Variable block size compressed RGBA Texture.	12x12: 0.89 bits per pixel (7744 bytes for a 256x256 Texture) 10x10: 1.28 bits per pixel (10816 bytes for a 256x256 Textures) 8x8: 2 bits per pixel (16KB for a 256x256 Texture); 6x6: 3.56 bits per pixel (29584 bytes for a 256x256 Texture) 5x5: 5.12 bits per pixel (43264 bytes for a 256x256 Texture) 4x4: 8 bits per pixel (64KB for a 256x256 Texture)	tvOS (all), iOS (A8), Android (PowerVR 6XT, Mali T600 series, Adreno 400 series, Tegra K1).
RGB Compressed PVRTC 2 bits	High-compression RGB Texture. Low quality, but lower size, resulting in higher performance.	16KB (2 bits per pixel)	Android (PowerVR), iOS, tvOS.
RGBA Compressed PVRTC 2 bits	High-compression RGBA Texture. Low quality, but lower size, resulting in higher performance.	16KB (2 bits per pixel)	Android (PowerVR), iOS, tvOS.

Texture compression format	Description	Size for a 256x256 pixel Texture	Platform Support
RGB Compressed PVRTC 4 bits	Compressed RGB Texture. High-quality Textures, particularly on color data, but can take a long time to compress.	32KB (4 bits per pixel)	Android (PowerVR), iOS, tvOS.
RGBA Compressed PVRTC 4 bits	Compressed RGB Texture. High-quality Textures, particularly on color data, but can take a long time to compress.	32KB (4 bits per pixel)	Android (PowerVR), iOS, tvOS.
RGB Compressed **ATC**	Compressed RGB Texture.	32KB (4 bits per pixel)	Android (Qualcomm - Adreno), iOS, tvOS.
RGBA Compressed ATC	Compressed RGBA Texture.	64KB (8 bits per pixel)	Android (Qualcomm - Adreno), iOS, tvOS.
RGB 16 bit	65 thousand colors with no alpha. Uses more memory than the compressed formats, but could be more suitable for **UI** or crisp Textures without gradients.	128KB (16 bits per pixel)	All platforms.
RGB 24 bit	True color, but without alpha.	192KB (24 bits per pixel)	All platforms
Alpha 8	High-quality alpha channel, but without any color.	64KB (8 bits per pixel)	All platforms.
RGBA 16 bit	Low-quality true color. This is the default compression for Textures that have an alpha channel.	128KB (16 bits per pixel)	All platforms.
RGBA 32 bit	True color with alpha. This is the highest quality compression for Textures that have an alpha channel.	256KB (32 bits per pixel)	All platforms.

You can import Textures from DDS files, but only DXT, BC compressed formats, or uncompressed pixel formats are supported.

Notes on Android

Unless you're targeting specific hardware (such as Tegra), ETC2 compression is the most efficient option for Android, offering the best balance of quality and file size (with associated memory size requirements). If you need an alpha channel, you could store it externally and still benefit from a lower Texture file size.

You can use ETC1 for Textures that have an alpha channel, but only if the build is for Android and the Textures are placed on an atlas (by specifying the packing tag). To enable this, tick the **Compress using ETC1** checkbox for the Texture. Unity splits the resulting atlas into two Textures, each without an alpha channel, and then combines them in the final parts of the render pipeline.

To store an alpha channel in a Texture, use RGBA16 bit compression, which is supported by all hardware vendors.

2017–11–20 Page amended with limited editorial review

Linear rendering on WebGL added in 2017.2

Crunch compression format updated in 2017.3

Tizen and Samsung TV support discontinued in 2017.3

Render Texture

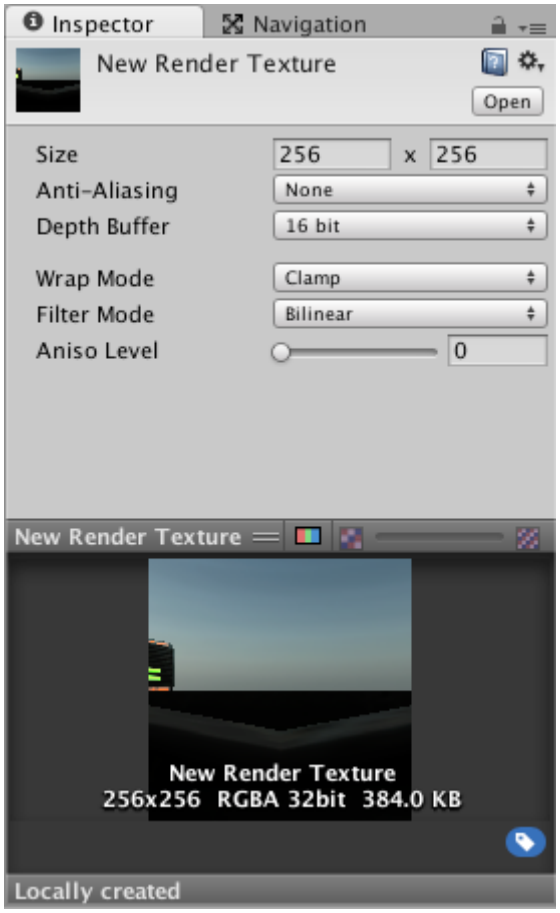
Leave feedback

SWITCH TO SCRIPTING

Render Textures are special types of **Textures** that are created and updated at runtime. To use them, you first create a new Render Texture and designate one of your Cameras to render into it. Then you can use the Render Texture in a **Material** just like a regular Texture. The Water **prefabs** in Unity **Standard Assets** are an example of real-world use of Render Textures for making real-time reflections and refractions.

Properties

The Render Texture **Inspector** is different from most Inspectors, but very similar to the Texture Inspector.



The Render Texture Inspector is almost identical to the Texture Inspector

The Render Texture inspector displays the current contents of Render Texture in realtime and can be an invaluable debugging tool for effects that use render textures.

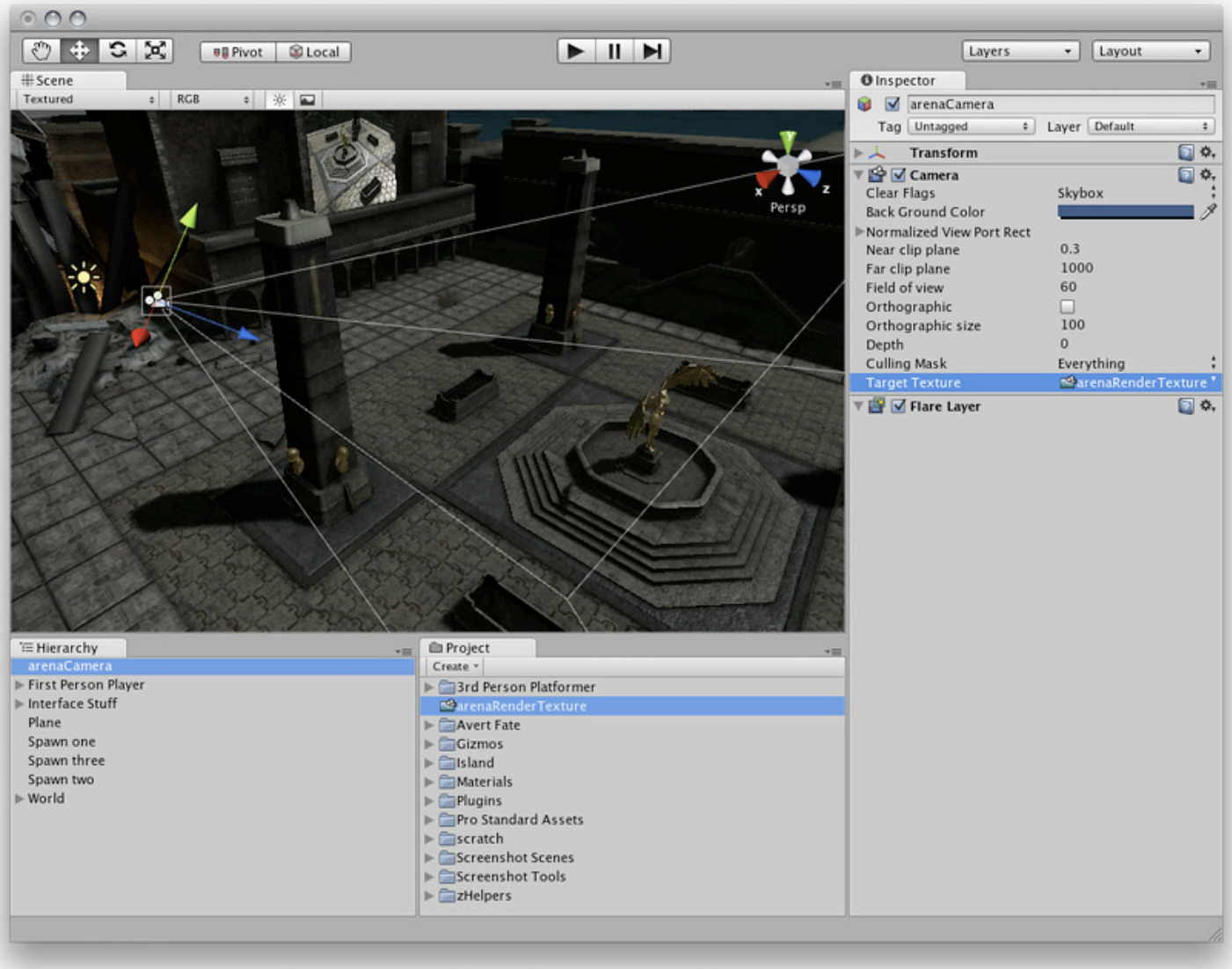
Property:	*Function:*
Size	The size of the Render Texture in **pixels**. Observe that only power-of-two values sizes can be chosen.
Anti-Aliasing	The amount of anti-aliasing to be applied. None, two, four or eight samples.
Depth Buffer	The type of the depth buffer. None, 16 bit or 24 bit.
Wrap Mode	Selects how the Texture behaves when tiled:
Repeat	The Texture repeats (tiles) itself

Property:	*Function:*
Clamp	The Texture's edges get stretched
Filter Mode	Selects how the Texture is filtered when it gets stretched by 3D transformations:
No Filtering	The Texture becomes blocky up close
Bilinear	The Texture becomes blurry up close
Trilinear	Like Bilinear, but the Texture also blurs between the different mip levels
Aniso Level	Increases Texture quality when viewing the texture at a steep angle. Good for floor and ground textures

Example

A very quick way to make a live arena-camera in your game:

Create a new Render Texture asset using **Assets >Create >Render Texture**.
Create a new Camera using **GameObject > Camera**.
Assign the Render Texture to the **Target Texture** of the new Camera.
Create a wide, tall and thin box
Drag the Render Texture onto it to create a Material that uses the render texture.
Enter Play Mode, and observe that the box's texture is updated in real-time based on the new Camera's output.



Render Textures are set up as demonstrated above

2017–09–19 Page amended with limited editorial review

GameObject menu changed in Unity 4.6

Custom Render Textures

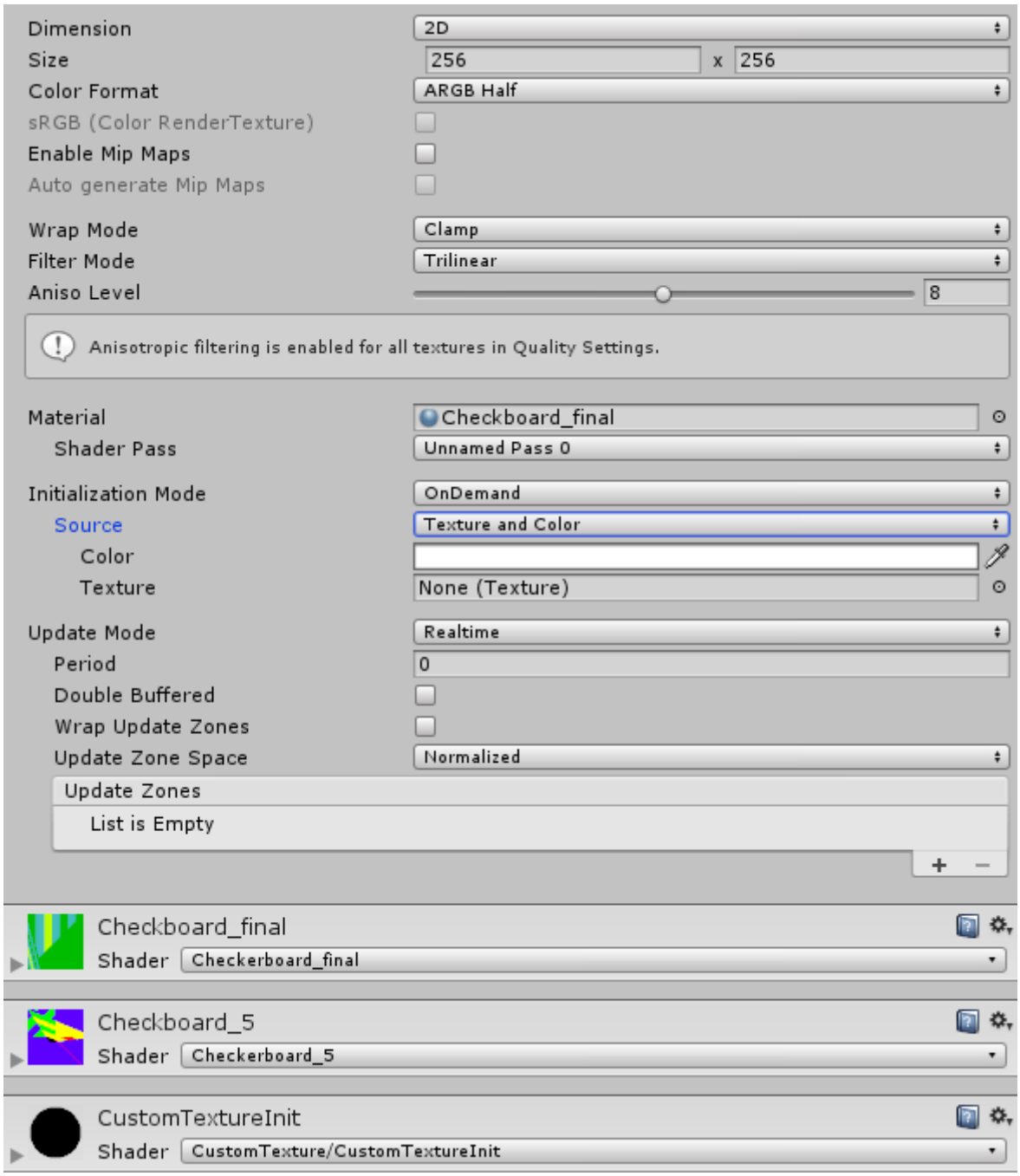
Leave feedback

SWITCH TO SCRIPTING

Custom **Render Textures** are an extension to **Render Textures** that allows users to easily update said texture with a **shader**. This is useful to implement all kind of complex simulations like caustics, ripple simulation for rain effects, splatting liquids against a wall, **etc**. It also provides a scripting and **Shader** framework to help with more complicated configuration like partial or multi-pass updates, varying update frequency **etc**.

To use them, you need to create a new Custom Render Texture **asset** and assign a Material to it. This Material will then update the content of the texture according to its various parameters. The Custom Render Texture can then be assigned to any kind of Material just like a regular texture, even one used for another Custom Render Texture.

Properties



The **inspector** for Custom Render Textures will display most of the properties of the Render Texture **inspector** as well as many specific ones.

Render Texture:

Property:	Function:
Dimension	Dimension of the Render Texture
2D	Render Texture will be two dimensional.
Cube	Render Texture will be a cube map
3D	Render Texture will be three dimensional
Size	The size of the Render Texture in **pixels**.
Color Format	Format of the Render Texture
sRGB (Color Render Texture)	Does this render texture use sRGB read/write conversions (Read Only).

Property:	Function:
Enable Mip Maps	Does this render texture use Mip Maps?
Auto generate Mip Maps	Enable to automatically generate Mip Maps.
Wrap Mode	Selects how the Texture behaves when tiled
Repeat	The Texture repeats (tiles) itself
Clamp	The Texture's edges get stretched
Filter Mode	Selects how the Texture is filtered when it gets stretched by 3D transformations:
Point	The Texture becomes blocky up close
Bilinear	The Texture becomes blurry up close
Trilinear	Like Bilinear, but the Texture also blurs between the different mip levels
Aniso Level	Increases Texture quality when viewing the texture at a steep angle. Good for floor and ground textures

Custom Texture:

Custom Texture parameters are separated in three main categories:

Material: Defines what shader is used to update the texture.

Initialization: Controls how the texture is initialized before any update is done by the shader

Update: Controls how the texture is updated by the shader.

Property:	Function:
Material	Material used to update the Custom Render Texture
Shader Pass	Shader Pass used to update the Custom Texture. The combo box will show all passes available in your Material.
Initialization Mode	Rate at which the texture should be initialized.
OnLoad	The texture is initialized once upon creation.
Realtime	The texture is initialized every frame.
OnDemand	The texture is initialized on demand from the script.
Source	How the texture should be initialized.
Texture and Color	The texture will be initialized by a texture multiplied by a color.
Initialization Color	Color with which the custom texture is initialized. If an initialization texture is provided as well, the custom texture will be initialized by the multiplication of the color and the texture.
Initialization Texture	Texture with which the custom texture is initialized. If an initialization color is provided as well, the custom texture will be initialized by the multiplication of the color and the texture.
Material	The texture will be initialized by a material.

Property:	Function:
Initialization Material	Material with which the custom texture is initialized.
Update Mode	Rate at which the texture should be updated by the shader.
OnLoad	The texture is updated once upon creation.
Realtime	The texture is updated every frame.
OnDemand	The texture is updated on demand from script.
Period	(Realtime only) Period in seconds at which real-time texture is updated (0.0 will update every frame).
Double Buffered	The texture will be double buffered. Each update will swap the two buffers, allowing user to read the result of the preceding update in the shader.
Wrap Update Zones	Enable to allow partial update zones to wrap around the border of the texture.
Cubemap Faces	(Cubemap only) Series of toggle allowing user to enable/disable update on each of the **cubemap** faces.
Update Zone Space	Coordinate system in which update zones are defined.
Normalized	All coordinates and sizes are between 0 and 1 with the top-left corner starting at (0, 0)
Pixel	All coordinates and sizes are expressed in pixels limited by the width and height of the texture. Top-left corner starting at (0, 0)
Update Zone List	List of update zones for the texture (see below for more details)

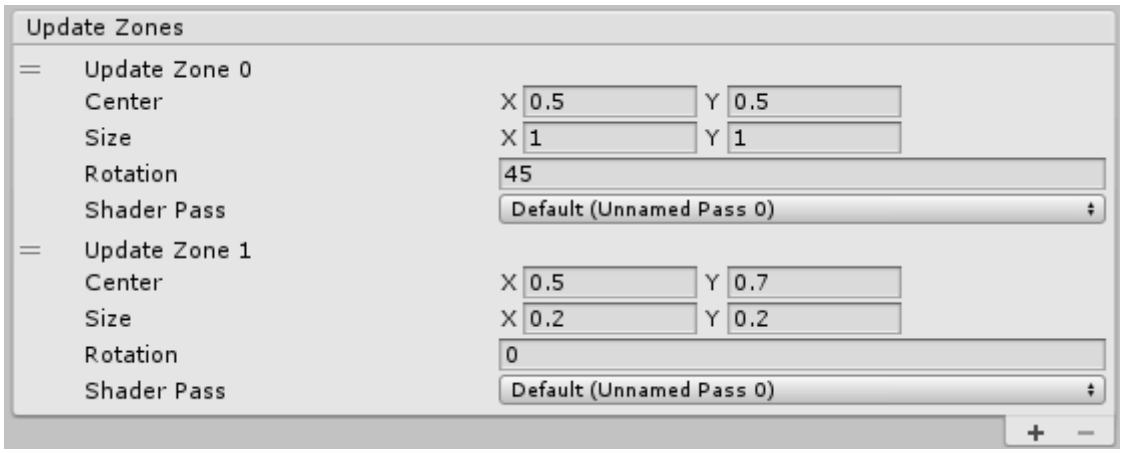
Exporting Custom Render Texture to a file:

Custom Render Textures can be exported to a PNG or EXR file (depending on the texture format) via the contextual “Export” menu.

Update Zones:

By default, when the Custom Render Texture is updated, the whole texture is updated at once by the Material. One of the important features of the Custom Texture is the ability for the user to define zones of partial update. With this, users can define as many zones as they want and the order in which they are processed.

This can be used for several different purpose. For example, you could have multiple small zones to splat water drops on the texture and then do a full pass to simulate the ripples. This can also be used as an optimization when you know that you don’t need to update the full texture.



Update zones have their own set of properties. The **Update Zone Space** will be reflected in the display. Depending on the **Dimension** of the texture, coordinates will be 2D (for 2D and Cube textures) or 3D (for 3D textures).

Property:	Function:
Center	Coordinate of the center of the update zone.
Size	Size of the update zone.
Rotation	Orientation of the update zone in degrees (unavailable for 3D textures).
Shader Pass	Shader Pass to use for this update zone. If left as default, this update zone will use the Shader Pass defined in the main part of the inspector otherwise it will use the provided one.
Swap (Double Buffer)	(Only for Double Buffered textures) If this is true, the buffers will be swapped before this update zone is processed.

Double Buffered Custom Textures

Custom Render Textures can be “Double Buffered”. Internally, there are two textures but from the user standpoint they are the same. After each update, the two textures will be swapped. This allows the user to read the result of the last update while writing a new result in the Custom Render Texture. This is particularly useful if the shader needs to use the content already written in the texture but cannot mix the values with classic blend modes. This is also needed if the shaders has to sample different pixels of the preceding result.

Performance Warning: Due to some technicalities, the double buffering currently involves a copy of the texture at each swap which can lead to a drop in performance depending on the frequency at which it is done and the resolution of the texture.

Chaining Custom Render Textures

Custom Render Textures need a Material to be updated. This Material can have textures as input. This means that the a Custom Texture can be used as an input to generate another one. This way, users can chain up several Custom Textures to generate a more complex multi-step simulation. The system will correctly handle all the dependencies so that the different updates happen in the right order.

Writing a shader for a Custom Render Texture

Updating a Custom Texture is like doing a 2D post process in a Render Texture. To help users write their custom texture shaders, we provide a small framework with utility functions and built-in helper variables.

Here is a really simple example that will fill the texture with a color multiplied by a color:

```
Shader "CustomRenderTexture/Simple"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
                                _Tex("InputTex", 2D) = "white" {}
     }

     SubShader
     {
        Lighting Off
        Blend One Zero

        Pass
        {
            CGPROGRAM
            #include "UnityCustomRenderTexture.cginc"
            #pragma vertex CustomRenderTextureVertexShader
            #pragma fragment frag
             #pragma target 3.0

            float4      _Color;
                                sampler2D   _Tex;

            float4 frag(v2f_customrendertexture IN) : COLOR
            {
                return _Color * tex2D(_Tex, IN.localTexcoord.xy);
            }
            ENDCG
                    }
    }
}
```

The only mandatory steps when writing a shader for a custom texture are:

* #include “UnityCustomRenderTexture.cginc”

Use the provided **Vertex Shader** *CustomRenderTextureVertexShader*

Use the provided input structure *v2f_customrendertexture* for the pixel shader

Other than that, the user is free to write the pixel shader as he wishes.

Here is another example for a shader used in an initialization material:

```
Shader "CustomRenderTexture/CustomTextureInit"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _Tex("InputTex", 2D) = "white" {}
    }

    SubShader
    {
        Lighting Off
        Blend One Zero

        Pass
        {
            CGPROGRAM
            #include "UnityCustomRenderTexture.cginc"

            #pragma vertex InitCustomRenderTextureVertexShader
            #pragma fragment frag
            #pragma target 3.0

            float4      _Color;
                                        sampler2D   _Tex;

            float4 frag(v2f_init_customrendertexture IN) : COLOR
            {
                _Color * tex2D(_Tex, IN.texcoord.xy);
            }
            ENDCG
        }
    }
}
```

Same as for the update shader, the only mandatory steps are these:

* #include “UnityCustomRenderTexture.cginc”

Use the provided Vertex Shader Init*CustomRenderTextureVertexShader*

Use the provided input structure *v2f_init_customrendertexture* for the pixel shader

In order to help the user in this process we provide a set of built-in values:

Input values from the ***v2f_customrendertexture*** structure:

Name	Type	Value
`localTexcoord`	`float3`	Texture coordinates relative to the update zone being currently processed.
`globalTexcoord`	`float3`	Texture coordinates relative to the Custom Render Texture itself
`primitiveID`	`uint`	Index of the update zone being currently processed.
`direction`	`float3`	For Cube Custom Render Texture, direction of the current pixel inside the cubemap.

Input values from the ***v2f_init_customrendertexture*** structure:

Name	Type	Value
`texcoord`	float3	Texture coordinates relative to the Custom Render Texture itself.

Global values:

Name	Type	Value
`_CustomRenderTextureWidth`	`float`	Width of the Custom Texture in pixels
`_CustomRenderTextureHeight`	`float`	Height of the Custom Texture in pixels
`_CustomRenderTextureDepth`	`float`	Depth of the Custom Texture in pixels (only for 3D textures, otherwise will always be equal to 1).
`_CustomRenderTextureCubeFace`	`float`	Only for Cubemaps: Index of the current cubemap face being processed (-X, +X, -Y, +Y, -Z, +Z).
`_CustomRenderTexture3DSlice`	`float`	Only for 3D textures: Index of the current 3D slice being processed.
`_SelfTexture2D`	`Sampler2D`	For double buffered textures: Texture containing the result of the last update before the last swap.
`_SelfTextureCube`	`SamplerCUBE`	For double buffered textures: Texture containing the result of the last update before the last swap.
`_SelfTexture3D`	`Sampler3D`	For double buffered textures: Texture containing the result of the last update before the last swap.

Controlling Custom Render Texture from Script

Most of the functionalities described here can be accessed via the Scripting API. Changing material parameters, update frequency, update zones, requesting an update etc, can all be done with the script.

One thing to keep in mind though is that any update requested for the Custom Texture will happen at a very specific time at the beginning of the frame with the current state of the Custom Texture. This guarantees that any material using this texture will have the up-to-date result.

This means that this kind of pattern:

```
customRenderTexture.updateZones = updateZones1;
customRenderTexture.Update();
customRenderTexture.updateZones = updateZones2;
customRenderTexture.Update();
```

Will not yield the “expected” result of one update done with the first array of update zones and then a second update with the other array. This will do two updates with the second array.

The good rule of thumb is that any property modified will only be active in the next frame.

2017–05–18 Page published with no editorial review

New feature in Unity 2017.1

Movie Textures

Leave feedback

SWITCH TO SCRIPTING

Note: MovieTexture is due to be deprecated in a future version of Unity. You should use VideoPlayer for video download and movie playback.

Movie Textures are animated **Textures** that are created from a video file. By placing a video file in your project's **Assets Folder**, you can import the video to be used exactly as you would use a regular Texture.

Video files are imported via Apple QuickTime. Supported file types are what your QuickTime installation can play (usually **.mov**, **.mpg**, **.mpeg**, **.mp4**, **.avi**, **.asf**). On Windows, movie importing requires Quicktime to be installed. Download Quicktime from Apple Support Downloads.

Properties

The Movie Texture **Inspector** is very similar to the regular Texture Inspector.

Video files are Movie Textures in Unity

Property:	*Function:*
Aniso Level	Increases Texture quality when viewing the texture at a steep angle. Good for floor and ground textures
Filtering Mode	Selects how the Texture is filtered when it gets stretched by 3D transformations

Property:	*Function:*
Loop	If enabled, the movie will loop when it finishes playing
Quality	**Compression** of the Ogg Theora video file. A higher value means higher quality, but larger file size

Details

When a video file is added to your Project, it will automatically be imported and converted to **Ogg Theora** format. Once your Movie Texture has been imported, you can attach it to any **GameObject** or **Material**, just like a regular Texture.

Playing the Movie

Your Movie Texture will not play automatically when the game begins running. You must use a short script to tell it when to play.

```
// this line of code will make the Movie Texture begin playing
((MovieTexture)GetComponent<Renderer>().material.mainTexture).Play();
```

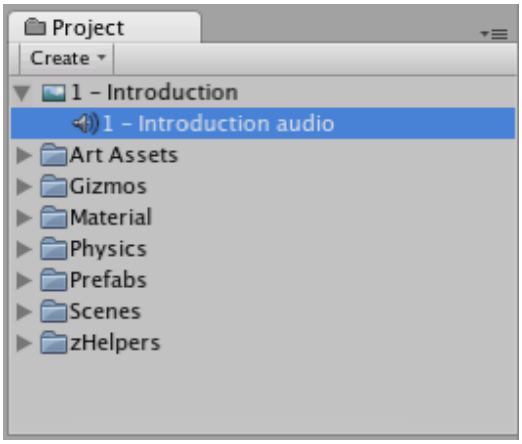
Attach the following script to toggle Movie playback when the space bar is pressed:

```
public class PlayMovieOnSpace : MonoBehaviour {
    void Update () {
        if (Input.GetButtonDown ("Jump")) {

            Renderer r = GetComponent<Renderer>();
            MovieTexture movie = (MovieTexture)r.material.mainTexture;

            if (movie.isPlaying) {
                movie.Pause();
            }
            else {
                movie.Play();
            }
        }
    }
}
```

For more information about playing Movie Textures, see the Movie Texture Script Reference page

Movie Audio

When a Movie Texture is imported, the audio track accompanying the visuals are imported as well. This audio appears as an **AudioClip** child of the Movie Texture.



The video's audio track appears as a child of the Movie Texture in the Project View

To play this audio, the **Audio Clip** must be attached to a GameObject, like any other **Audio Clip**. Drag the **Audio Clip** from the **Project View** onto any GameObject in the **Scene** or Hierarchy View. Usually, this will be the same GameObject that is showing the Movie. Then use audio.Play() to make the the movie's audio track play along with its video.

iOS

Movie Textures are not supported on **iOS**. Instead, full-screen streaming playback is provided using Handheld.PlayFullScreenMovie.

You need to keep your videos inside the *StreamingAssets* folder located in the *Assets* folder of your project.

Unity iOS supports any movie file types that play correctly on an iOS device, implying files with the extensions **.mov**, **.mp4**, **.mpv**, and **.3gp** and using one of the following compression standards:

H.264 Baseline Profile Level 3.0 video
MPEG–4 Part 2 video

For more information about supported compression standards, consult the iPhone SDK MPMoviePlayerController Class Reference.

As soon as you call Handheld.PlayFullScreenMovie the screen will fade from your current content to the designated background color. It might take some time before the movie is ready to play but in the meantime, the player will continue displaying the background color and may also display a progress indicator to let the user know the movie is loading. When playback finishes, the screen will fade back to your content.

The video player does not respect switching to mute while playing videos

As written above, video files are played using Apple's embedded player (as of SDK 3.2 and iPhone OS 3.1.2 and earlier). This contains a bug that prevents Unity switching to mute.

The video player does not respect the device's orientation

The Apple video player and iPhone SDK do not provide a way to adjust the orientation of the video. A common approach is to manually create two copies of each movie in landscape and portrait orientations. Then, the orientation of the device can be determined before playback so the right version of the movie can be chosen.

Android

Movie Textures are not supported on Android. Instead, full-screen streaming playback is provided using Handheld.PlayFullScreenMovie.

You need to keep your videos inside the *StreamingAssets* folder located in the *Assets* folder of your project.

Unity Android supports any movie file type supported by Android, (ie, files with the extensions **.mp4** and **.3gp**) and using one of the following compression standards:

H.263
H.264 AVC
MPEG–4 SP
However, device vendors are keen on expanding this list, so some Android devices are able to play formats other than those listed, such as HD videos.

For more information about the supported compression standards, consult the Android SDK Core Media Formats documentation.

As soon as you call Handheld.PlayFullScreenMovie the screen will fade from your current content to the designated background color. It might take some time before the movie is ready to play but in the meantime, the player will continue displaying the background color and may also display a progress indicator to let the user know the movie is loading. When playback finishes, the screen will fade back to your content.

3D Textures

Leave feedback

SWITCH TO SCRIPTING

3D Textures can only be created from script. The following snippet creates 3D texture where each axis X, Y and Z corresponds to the colour values red, green and blue.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Example : MonoBehaviour
{

    Texture3D texture;

    void Start ()
    {
        texture = CreateTexture3D (256);
    }

    Texture3D CreateTexture3D (int size)
    {
        Color[] colorArray = new Color[size * size * size];
        texture = new Texture3D (size, size, size, TextureFormat.RGBA32, true);
        float r = 1.0f / (size - 1.0f);
        for (int x = 0; x < size; x++) {
            for (int y = 0; y < size; y++) {
                for (int z = 0; z < size; z++) {
                    Color c = new Color (x * r, y * r, z * r, 1.0f);
                    colorArray[x + (y * size) + (z * size * size)] = c;
                }
            }
        }
        texture.SetPixels (colorArray);
        texture.Apply ();
        return texture;
    }

}
```

Texture arrays

Leave feedback

SWITCH TO SCRIPTING

See Advanced ShaderLab topics.

Rendering Components

Leave feedback

This group contains all **Components** that have to do with **rendering** in-game and user interface elements. Lighting and special effects are also included in this group.

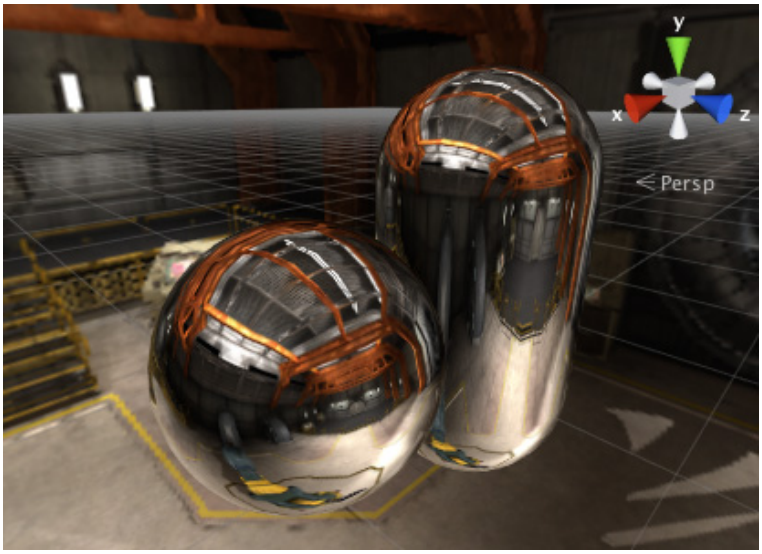
Cubemap

Leave feedback

SWITCH TO SCRIPTING

A **Cubemap** is a collection of six square textures that represent the reflections on an environment. The six squares form the faces of an imaginary cube that surrounds an object; each face represents the view along the directions of the world axes (up, down, left, right, forward and back).

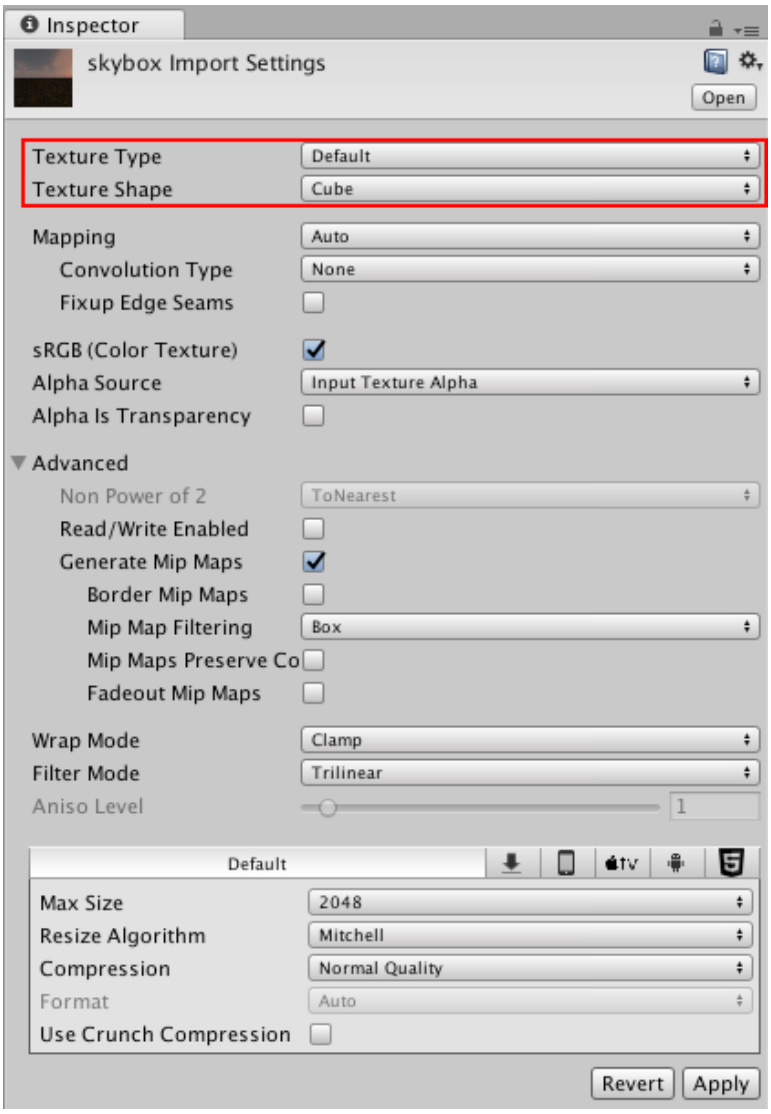
Cubemaps are often used to capture reflections or "surroundings" of objects; for example skyboxes and environment reflections often use cubemaps.



Cubemapped skybox and reflections

Creating Cubemaps from Textures

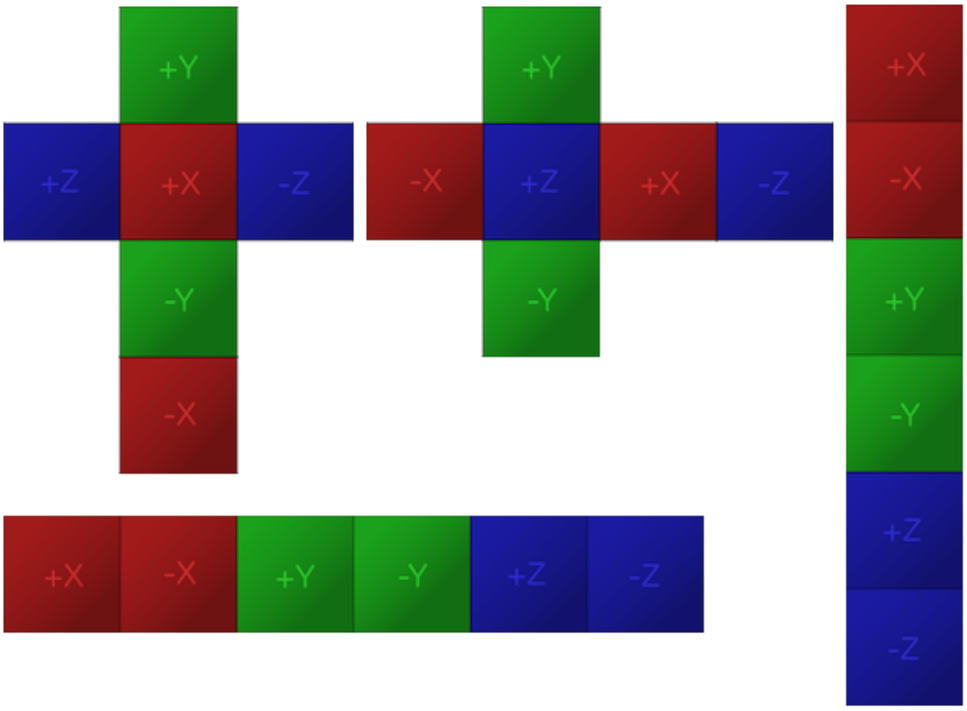
The fastest way to create cubemaps is to import them from specially laid out Textures. Select the Texture in the Project window, to see the Import Settings in the **Inspector** window. In the Import Settings, set the **Texture Type** to **Default**, **Normal Map** or **Single Channel**, and the **Texture Shape** to **Cube**. Unity then automatically sets the Texture up as a Cubemap.



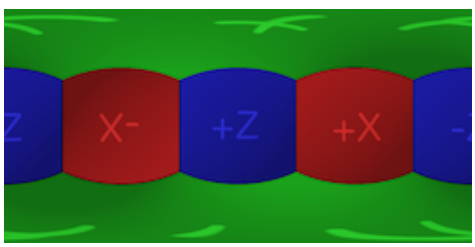
Cubemap texture import type

Several commonly-used cubemap layouts are supported (and in most cases, Unity detects them automatically).

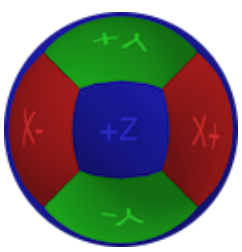
Vertical and horizontal cross layouts, as well as column and row of cubemap faces are supported:



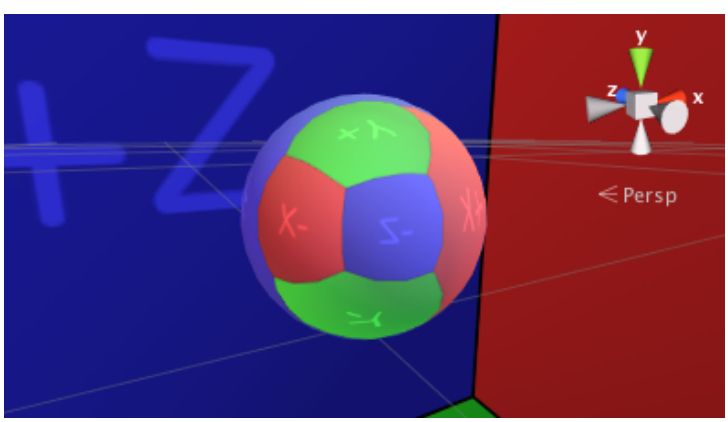
Another common layout is `LatLong` (Latitude-Longitude, sometimes called cylindrical). Panorama images are often in this layout:



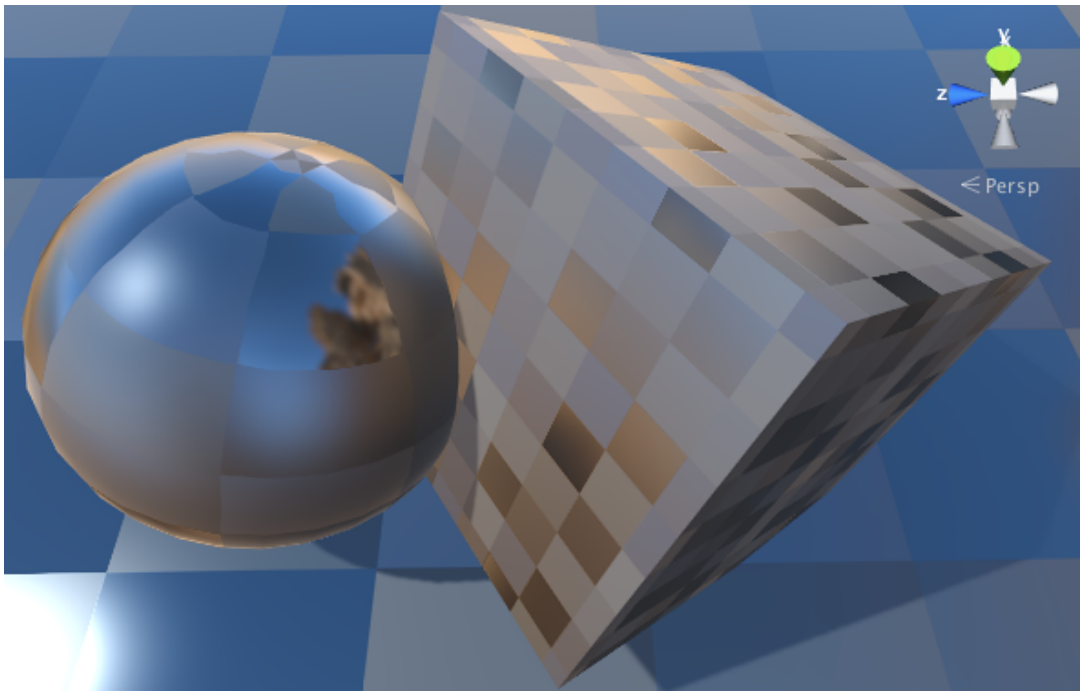
`SphereMap` (spherical environment map) images can also be found:



By default Unity looks at the aspect ratio of the imported texture to determine the most appopriate layout from the above. When imported, a cubemap is produced which can be used for skyboxes and reflections:



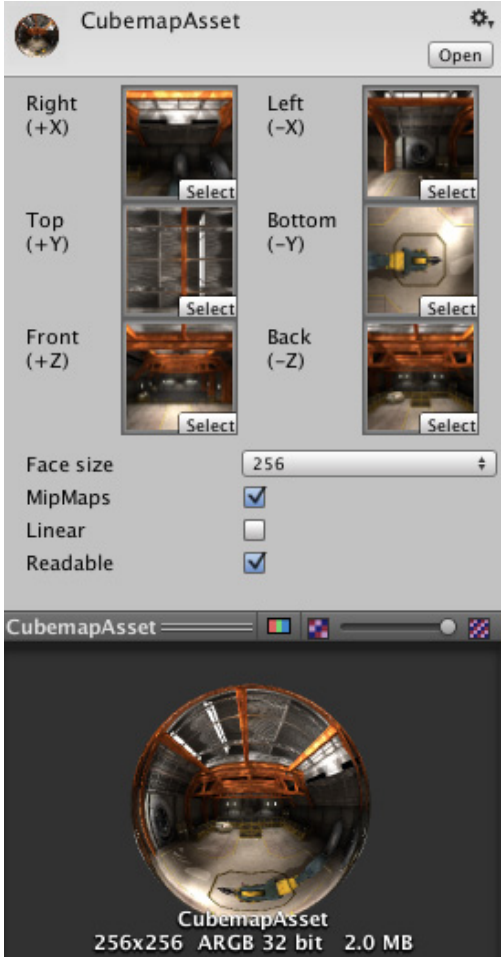
Selecting `Glossy Reflection` option is useful for cubemap textures that will be used by Reflection Probes. It processed cubemap mip levels in a special way (specular convolution) that can be used to simulate reflections from surfaces of different smoothness:



Cubemap used in a Reflection Probe on varying-smoothness surface

Legacy Cubemap Assets

Unity also supports creating cubemaps out of six separate textures. Select **Assets > Create > Legacy > Cubemap** from the menu, and drag six textures into empty slots in the inspector.



Legacy Cubemap Inspector

Property:	*Function:*
Right..Back Slots	Textures for the corresponding cubemap face.
Face Size	Width and Height of each Cubemap face in pixels. The textures will be scaled automatically to fit this size.
Mipmap	Should mipmaps be created?
Linear	Should the cubemap use linear color?
Readable	Should the cubemap allow scripts to access the pixel data?

Note that it is preferred to create cubemaps using the Cubemap texture import type (see above) - this way cubemap texture data can be compressed; edge fixups and glossy reflection convolution be performed; and HDR cubemaps are supported.

Other Techniques

Another useful technique is to generate the cubemap from the contents of a Unity scene using a script. The Camera.RenderToCubemap function can record the six face images from any desired position in the scene; the code example on the function's script reference page adds a menu command to make this task easy.

2018–01–31 Page amended with limited editorial review

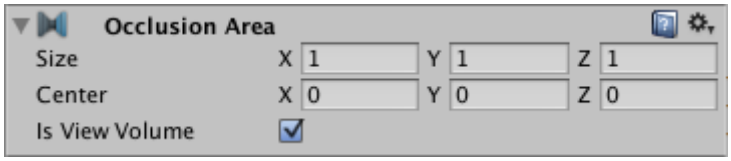
Occlusion Area

Leave feedback

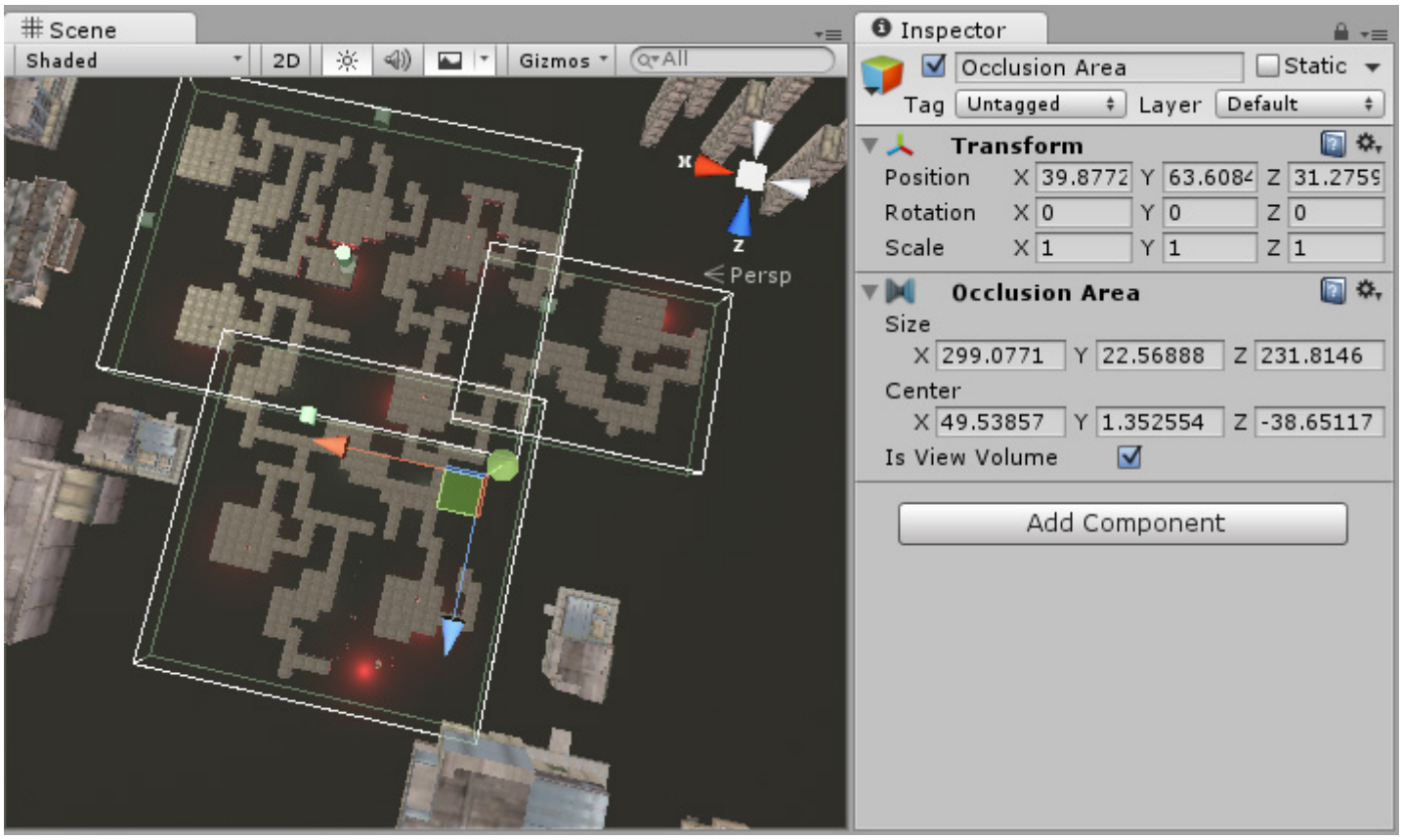
SWITCH TO SCRIPTING

To apply occlusion culling to moving objects you have to create an **Occlusion Area** and then modify its size to fit the space where the moving objects will be located (of course the moving objects cannot be marked as static). You can create Occlusion Areas by adding the **Occlusion Area** component to an empty game object (**Component -> Rendering -> Occlusion Area** in the menus).

After creating the **Occlusion Area**, check the *Is View Volume* checkbox to occlude moving objects.

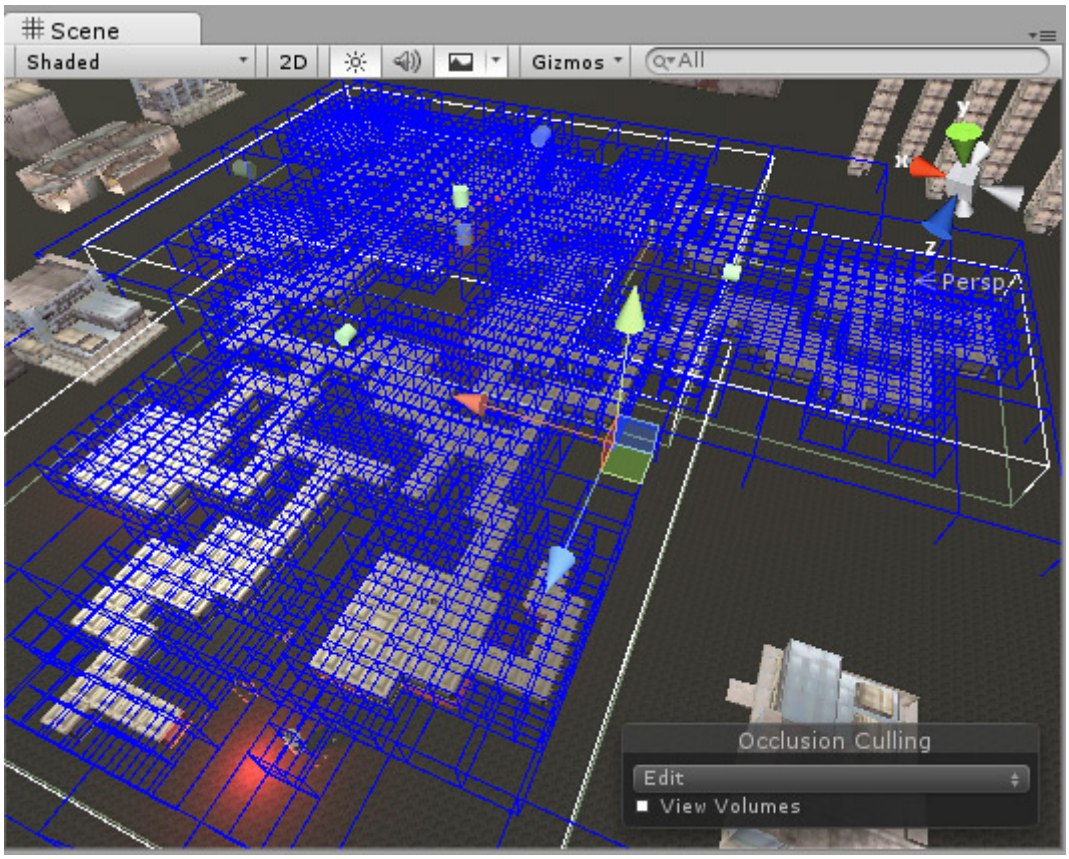


Property:	*Function:*
Size	Defines the size of the Occlusion Area.
Center	Sets the center of the Occlusion Area. By default this is 0,0,0 and is located in the center of the box.
Is View Volume	Defines where the **camera** can be. Check this in order to occlude static objects that are inside this *Occlusion Area*.



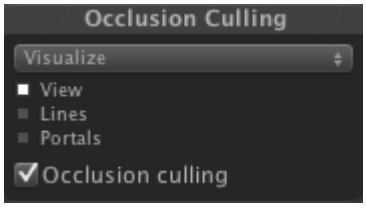
Occlusion Area properties for moving objects.

After you have added the Occlusion Area, you need to see how it divides the box into cells. To see how the occlusion area will be calculated, select **Edit** and toggle the **View** button in the **Occlusion Culling Preview Panel**.



Testing the generated occlusion

After your occlusion is set up, you can test it by enabling the *Occlusion Culling* (in the **Occlusion Culling Preview Panel** in Visualize mode) and moving the **Main Camera** around in the **scene view**.



The Occlusion View mode in Scene View

As you move the Main Camera around (whether or not you are in Play mode), you'll see various objects disable themselves. The thing you are looking for here is any error in the occlusion data. You'll recognize an error if you see objects suddenly popping into view as you move around. If this happens, your options for fixing the error are either to change the resolution (if you are playing with target volumes) or to move objects around to cover up the error. To debug problems with occlusion, you can move the Main Camera to the problematic position for spot-checking.

When the processing is done, you should see some colorful cubes in the View Area. The blue cubes represent the cell divisions for **Target Volumes**. The white cubes represent cell divisions for **View Volumes**. If the parameters were set correctly you should see some objects not being rendered. This will be because they are either outside of the view frustum of the camera or else occluded from view by other objects.

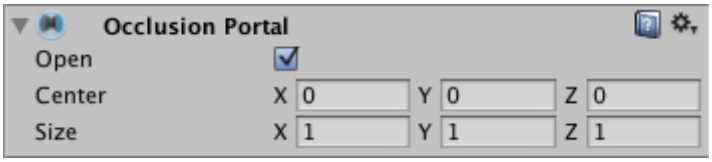
After occlusion is completed, if you don't see anything being occluded in your **scene** then try breaking your objects into smaller pieces so they can be completely contained inside the cells.

Occlusion Portals

Leave feedback

SWITCH TO SCRIPTING

In order to create occlusion primitives which are openable and closable at runtime, Unity uses **Occlusion Portals**.



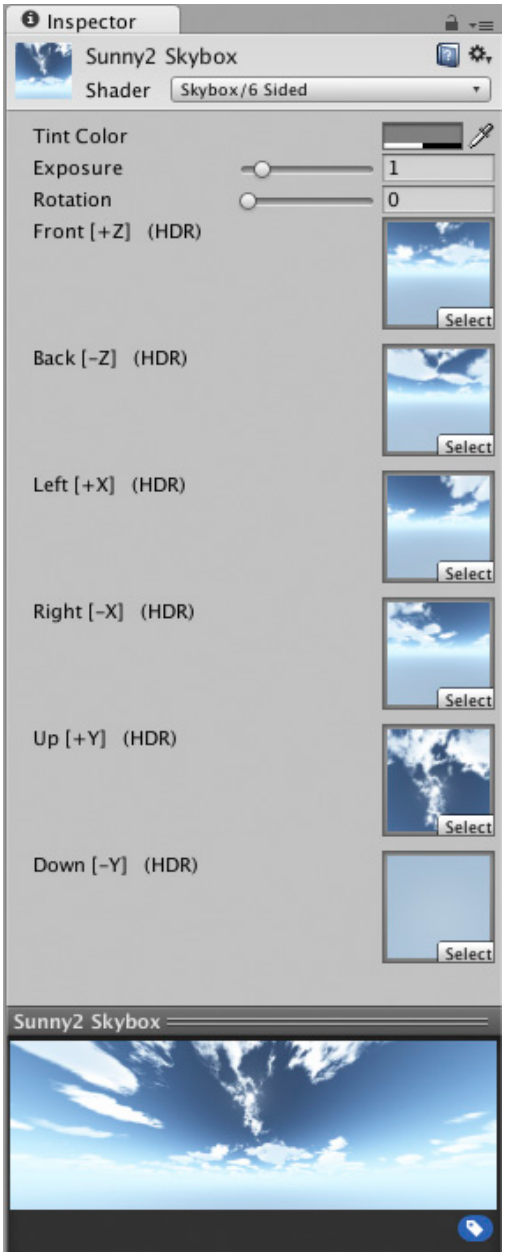
Property:	*Function:*
Open	Indicates if the portal is open (scriptable).
Center	Sets the center of the **Occlusion Area**. By default this is 0,0,0 and is located in the center of the box.
Size	Defines the size of the Occlusion Area.

Skybox

Leave feedback

SWITCH TO SCRIPTING

Skyboxes are a wrapper around your entire **scene** that shows what the world looks like beyond your geometry.



Properties

Property:	*Function:*
Tint Color	The tint color
Exposure	Adjusts the brightness of the **skybox**.
Rotation	Changes the rotation of the skybox around the positive y axis.
Front, etc	The textures used for each face of the cube used to store the skybox. Note that it is important to get these textures into the correct slot.

Details

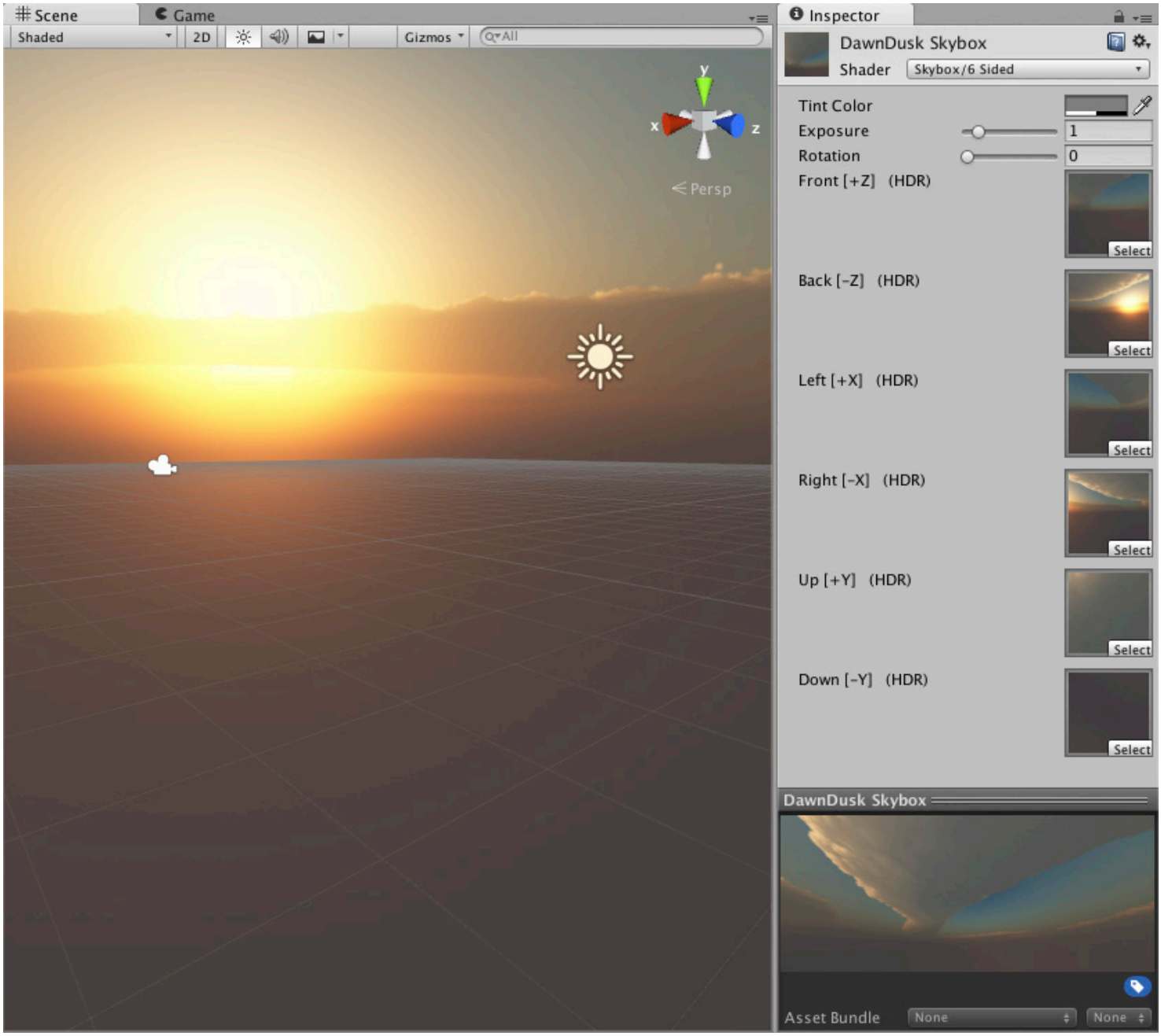
Skyboxes are rendered around the whole scene in order to give the impression of complex scenery at the horizon. Internally skyboxes are rendered after all opaque objects; and the **mesh** used to render them is either a box with six textures, or a

tessellated sphere.

To implement a Skybox create a skybox material. Then add it to the scene by using the **Window** > **Rendering** > **Lighting Settings** menu item and specifying your skybox material as the Skybox on the Scene tab.

Adding the Skybox **Component** to a Camera is useful if you want to override the default Skybox. E.g. You might have a split screen game using two Cameras, and want the Second camera to use a different Skybox. To add a Skybox Component to a Camera, click to highlight the Camera and go to **Component->Rendering->Skybox**.

If you want to create a new Skybox, use this guide.



Hints

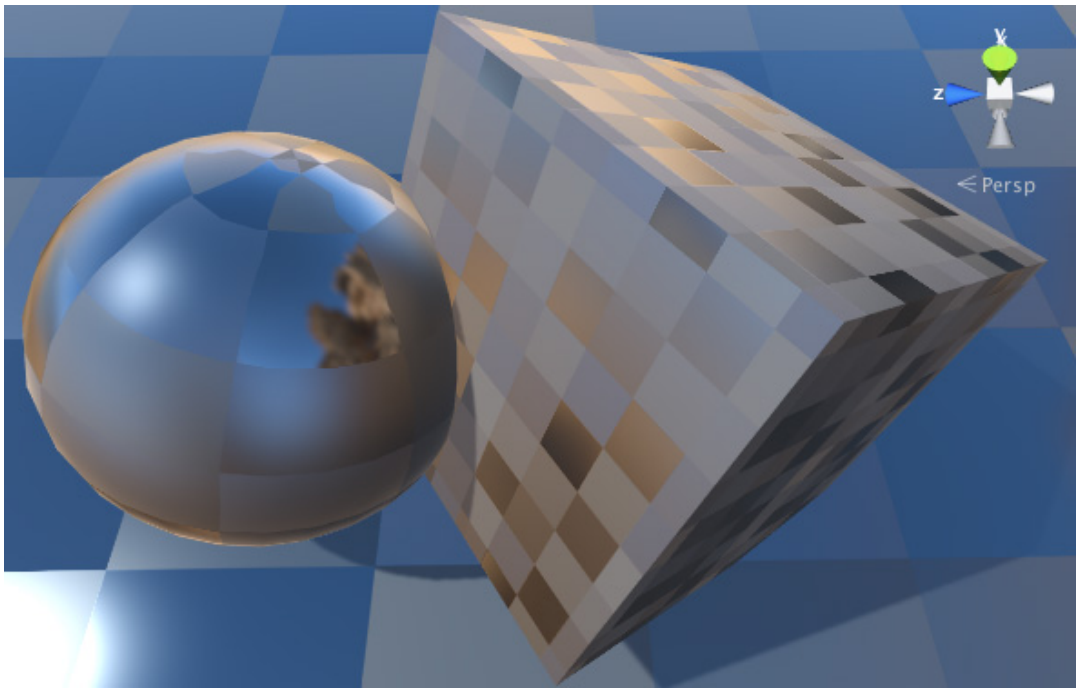
If you have a Skybox assigned to a **Camera**, make sure to set the **Camera**'s **Clear mode** to Skybox.
It's a good idea to match your Fog color to the color of the skybox. Fog color can be set in the Lighting window.

Reflection Probe

Leave feedback

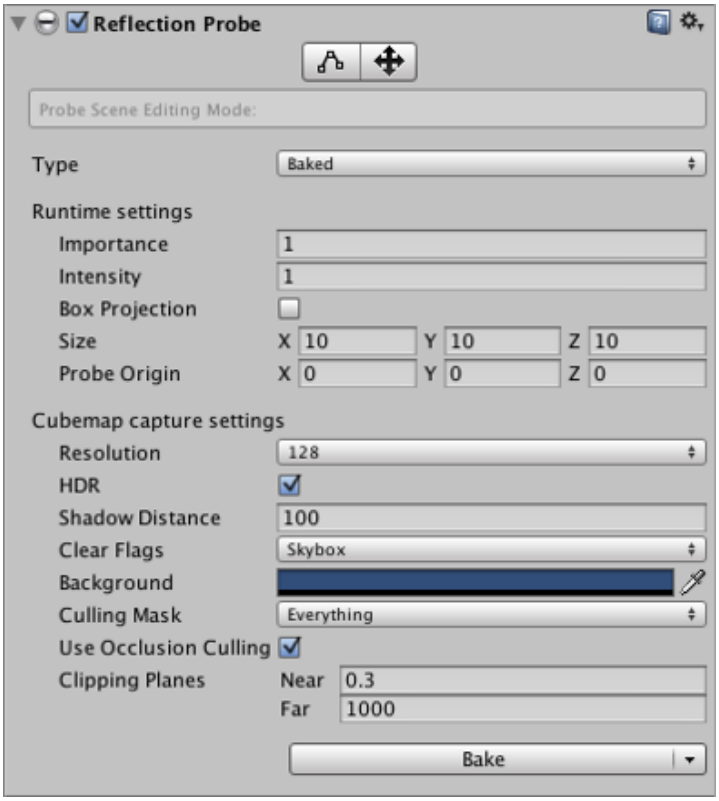
SWITCH TO SCRIPTING

A **Reflection Probe** is rather like a **camera** that captures a spherical view of its surroundings in all directions. The captured image is then stored as a Cubemap that can be used by objects with reflective materials. Several reflection probes can be used in a given **scene** and objects can be set to use the cubemap produced by the nearest probe. The result is that the reflections on the object can change convincingly according to its environment.



A Reflection Probe showing reflections from a nearby object

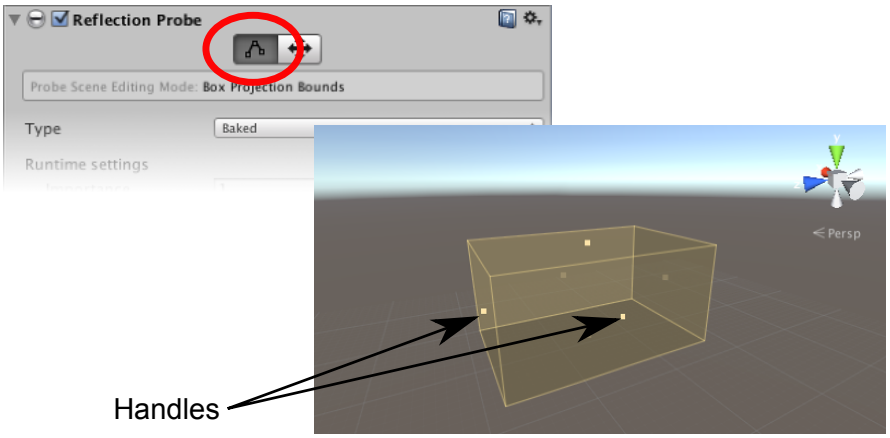
Properties



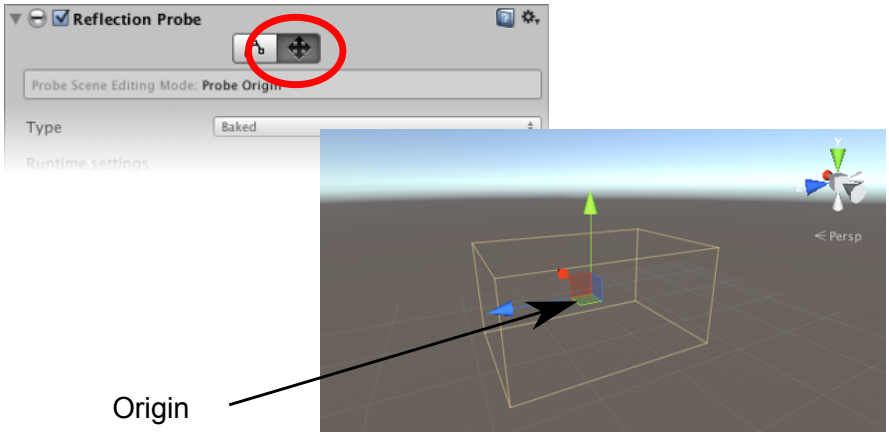
Property:	*Function:*
Type	Choose whether the probe is for a **Baked**, **Custom** or **Realtime** setup?
Dynamic Objects	(**Custom** type only) Forces objects not marked as **Static** to be baked in to the reflection.
Cubemap	(**Custom** type only) Sets a custom cubemap for the probe.
Refresh Mode	(**Realtime** type only) Selects if and how the probe will refresh at runtime. The *On Awake* option renders the probe only once when it first becomes active. *Every Frame* renders the probe every frame update, optionally using **Time Slicing** (see below). The **Via Scripting** option refreshes the probe from a user script command rather than an automatic update.
Time Slicing	(**Realtime** type only) How should the probe distribute its updates over time? The options are **All Faces At Once** (spreads update over nine frames), **Individual Faces** (updates over fourteen frames) and **No Time Slicing** (the update happens entirely within one frame). See below for further details.
Runtime settings	
Importance	The degree of "importance" of this probe compared to its neighbours. Higher values indicate greater importance; more important probes will have priority over less important one in cases where an object is within range of two or more probes. This setting also affects the **Blending**, explained here.
Intensity	The intensity modifier that is applied to the texture of this probe in its **shader**.
Box Projection	Check this box to enable projection for reflection UV mappings.
Box Size	The size of the box in which the reflection will be applied to the **GameObject**. The value is not affected by the Transform of the **GameObject**. Also used by **Box Projection**.
Box Offset	The center of the box in which the reflections will be applied to the GameObject. The value is relative to the position of the GameObject. Also used by **Box Projection**.
Cubemap capture settings	
Resolution	The resolution of the captured reflection image.
HDR	Should High Dynamic Range **rendering** be enabled for the cubemap? This also determines whether probe data is saved in OpenEXR or PNG format.
Shadow Distance	Distance at which shadows are drawn when rendering the probe.
Clear Flags	Option to specify how empty background areas of the cubemap will be filled. The options are **Skybox** and **Solid Color**.
Background	Background colour to which the reflection cubemap is cleared before rendering.
Culling Mask	Allows objects on specified layers to be included or excluded in the reflection. See the section about the Camera's culling mask on the Layers page.
Use Occlusion Culling	Should **occlusion culling** be used when baking the probe?
Clipping Planes	Near and **far clipping planes** of the probe's "camera".

Details

There are two buttons at the top of the **Reflection Probe Inspector** window that are used for editing the **Size** and **Probe Origin** properties directly within the Scene. With the leftmost button (**Size**) selected, the probe's zone of effect is shown in the scene as a yellow box shape with handles to adjust the box's size.



The other button (**Origin**) allows you to drag the probe's origin relative to the box. Note that the origin handle resembles the Transform position handle but the two positions are not the same. Also, the rotation and scale operations are not available for the probe box.



The probe's **Type** property determines how the reflection data is created and updated:

Baked probes store a static reflection cubemap generated by baking in the editor.
Custom probes store a static cubemap which can either be generated by baking or set manually by the user.
Realtime probes update the cubemap at runtime and can therefore react to dynamic objects in the scene.
To make use of the reflection cubemap, an object must have the **Reflection Probes** option enabled on its Mesh Renderer and also be using a shader that supports reflection probes. When the object passes within the volume set by the probe's **Size** and **Probe Origin** properties, the probe's cubemap will be applied to the object.

You can also manually set which reflection probe to use for a particular object using the settings on the object's Mesh Renderer. To do this, select one of the options for the **Mesh** Renderer's **Reflection Probes** property (**Simple**, **Blend Probes** or **Blend Probes and Skybox**) and drag the chosen probe onto its **Anchor Override** property.

See the Reflection Probes section in the manual for further details about principles and usage.

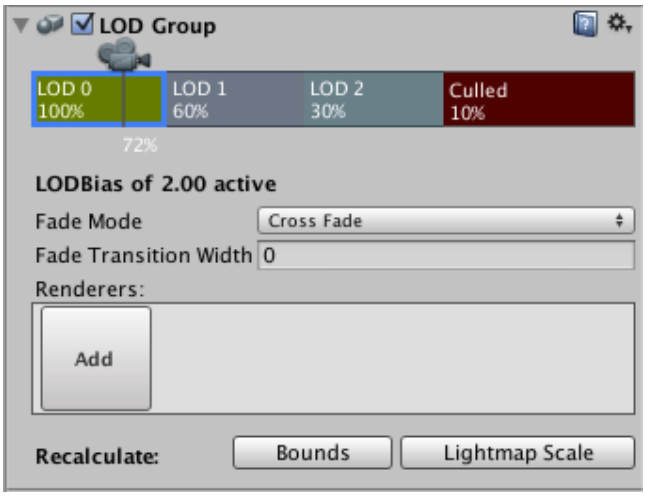
LOD Group

Leave feedback

SWITCH TO SCRIPTING

As your **scenes** get larger, performance becomes a bigger consideration. One of the ways to manage this is to have meshes with different levels of detail depending on how far the **camera** is from the object. This is called **Level of Detail** (abbreviated as **LOD**).

LOD Groups are used to manage *level of detail* (LOD) for **GameObjects**. Level of Detail is an optimisation technique that uses several meshes for an object; the meshes represent the same object with decreasing detail in the geometry. The idea is that the low-detail meshes are shown when the object is far from the camera and the difference will not be noticed. Since meshes with simpler geometry are less demanding on the hardware, performance can be improved by using LOD judiciously.



LOD Group inspector

Properties

The different LOD levels are visible in the horizontal bar with the camera icon just above it (LOD: 0, LOD: 1, LOD: 2, etc). The percentages in the LOD bars represent the fraction of the bounding box height relative to screen height where that LOD level becomes active. You can change the percentage values by dragging the vertical lines that separate the bars. You can also add and remove LOD levels from the bar by right-clicking it and selecting **Insert Before** or **Delete** from the contextual menu. The position of the camera icon along the bar shows the current percentage. The percentages in the LOD bars represent the thresholds at which the corresponding LOD level becomes active, measured by the ratio of the object's screen space height to screen height. Note that if the LOD Bias is not 1 the camera position is not necessarily the actual position where LOD transits from one to another.

When you click on one of the LOD bars to select it, a **Renderers** panel will be shown beneath. The "renderers" are actually GameObjects that hold the **mesh** to represent the LOD level; typically, this will be a child of the object that has the *LODGroup* **component**. If you click on an empty box (with the word "Add") in the Renderers panel, an object browser will appear to let you choose the object for that LOD level. Although you can choose *any* object for the renderer, you will be asked if you want to parent it to the LODGroup GameObject if it isn't already a child.

From Unity 5, you can choose Fade Mode for each LOD level. The fading is used to "blend" two neighboring LODs to achieve a smooth transition effect. However Unity doesn't provide a default built-in technique to blend LOD geometries. You need to implement your own technique according to your game type and **asset** production pipeline. Unity calculates a "blend factor" from the object's screen size and passes it to your **shader**.

There are two modes for calculating the blend factor:

Percentage: As the object's screen height goes from the current LOD height percentage to next, the blend factor goes from 1 to 0. Only the meshes of the current LOD will be rendered. Cross-fade: You need to specify a Transition Width value to define a cross-fading zone at the end of the current LOD where it will to transit to the next LOD. In the transition zone, both LOD levels will be rendered. The blend factor goes from 1 to 0 for the current LOD and 0 to 1 for the next LOD.

The *blend factor* is accessed as the `unity_LODFade.x uniform` variable in your shader program. Either keyword `LOD_FADE_PERCENTAGE` or `LOD_FADE_CROSSFADE` will be chosen for objects rendered with LOD fading.

For more details on naming conventions see the Level of detail page.

Look at the example of SpeedTree trees to see how *LODGroup* is configured and how the SpeedTree shader utilizes the `unity_LODFade` variable.

At the bottom of the **inspector** are two **Recalculate** buttons. The **Bounds** button will recalculate the **bounding volume** of the object after a new LOD level is added. The **Lightmap Scale** button updates the **Scale in Lightmap** property in the **lightmaps** based on changed LOD level boundaries.

Rendering Pipeline Details

Leave feedback

This section explains the technical details behind various aspects of Unity's **rendering** engine.

Deferred shading rendering path

Leave feedback

This page details the **deferred shading** rendering path. See Wikipedia: deferred shading for an introductory technical overview.

Overview

When using deferred shading, there is no limit on the number of lights that can affect a **GameObject**. All lights are evaluated per-pixel, which means that they all interact correctly with normal maps, **etc**. Additionally, all lights can have cookies and shadows.

Deferred shading has the advantage that the processing overhead of lighting is proportional to the number of **pixels** the light shines on. This is determined by the size of the light volume in the **Scene** regardless of how many GameObjects it illuminates. Therefore, performance can be improved by keeping lights small. Deferred shading also has highly consistent and predictable behaviour. The effect of each light is computed per-pixel, so there are no lighting computations that break down on large triangles.

On the downside, deferred shading has no real support for anti-aliasing and can't handle semi-transparent GameObjects (these are rendered using forward rendering). There is also no support for the **Mesh** Renderer's Receive Shadows flag and **culling masks** are only supported in a limited way. You can only use up to four **culling masks**. That is, your culling **layer mask** must at least contain all layers minus four arbitrary layers, so 28 of the 32 layers must be set. Otherwise you get graphical artefacts.

Requirements

It requires a graphics card with Multiple Render Targets (MRT), **Shader** Model 3.0 (or later) and support for Depth **render textures**. Most PC graphics cards made after 2006 support deferred shading, starting with GeForce 8xxx, Radeon X2400, Intel G45.

On mobile, deferred shading is supported on all devices running at least OpenGL ES 3.0.

Note: Deferred **rendering** is not supported when using Orthographic projection. If the **Camera**'s projection mode is set to Orthographic, the **Camera** falls back to Forward **rendering**.

Performance considerations

The rendering overhead of realtime lights in deferred shading is proportional to the number of pixels illuminated by the light and *not* dependent on Scene complexity. So small point or spot lights are very cheap to render and if they are fully or partially occluded by Scene GameObjects then they are even cheaper.

Of course, lights with shadows are much more expensive than lights without shadows. In deferred shading, shadow-casting GameObjects still need to be rendered once or more for each shadow-casting light. Furthermore, the lighting shader that applies shadows has a higher rendering overhead than the one used when shadows are disabled.

Implementation details

Objects with Shaders that do not support deferred shading are rendered after deferred shading is complete, using the forward rendering path.

The default layout of the render targets (RT0 - RT4) in the geometry buffer (g-buffer) is listed below. Data types are placed in the various channels of each render target. The channels used are shown in parentheses.

RT0, ARGB32 format: Diffuse color (RGB), occlusion (A).

RT1, ARGB32 format: Specular color (RGB), roughness (A).
RT2, ARGB2101010 format: World space normal (RGB), unused (A).
RT3, ARGB2101010 (non-HDR) or ARGBHalf (HDR) format: Emission + lighting + **lightmaps** + **reflection probes** buffer.
Depth+Stencil buffer.
So the default g-buffer layout is 160 bits/pixel (non-HDR) or 192 bits/pixel (HDR).

If using the Shadowmask or Distance Shadowmask modes for Mixed lighting, a fifth target is used:

RT4, ARGB32 format: Light occlusion values (RGBA).
And thus the g-buffer layout is 192 bits/pixel (non-HDR) or 224 bits/pixel (HDR).

If the hardware does not support five concurrent rendertargets then objects using shadowmasks will fallback to the forward rendering path. Emission+lighting buffer (RT3) is logarithmically encoded to provide greater dynamic range than is usually possible with an ARGB32 texture, when the Camera is not using **HDR**.

Note that when the Camera is using HDR rendering, there's no separate rendertarget being created for Emission+lighting buffer (RT3); instead the rendertarget that the Camera renders into (that is, the one that is passed to the image effects) is used as RT3.

G-Buffer pass

The g-buffer pass renders each GameObject once. Diffuse and specular colors, surface smoothness, world space normal, and emission+ambient+reflections+lightmaps are rendered into g-buffer textures. The g-buffer textures are setup as global shader properties for later access by shaders (*CameraGBufferTexture0 ..* CameraGBufferTexture3 names).

Lighting pass

The lighting pass computes lighting based on g-buffer and depth. Lighting is computed in screen space, so the time it takes to process is independent of Scene complexity. Lighting is added to the emission buffer.

Point and spot lights that do not cross the Camera's near plane are rendered as 3D shapes, with the Z buffer's test against the Scene enabled. This makes partially or fully occluded point and spot lights very cheap to render. Directional lights and point/spot lights that cross the near plane are rendered as fullscreen **quads**.

If a light has shadows enabled then they are also rendered and applied in this pass. Note that shadows do not come for "free"; shadow casters need to be rendered and a more complex light shader must be applied.

The only lighting model available is Standard. If a different model is wanted you can modify the lighting pass shader, by placing the modified version of the Internal-DeferredShading.shader file from the Built-in shaders into a folder named "Resources" in your "Assets" folder. Then go to the Edit->Project Settings->Graphics window. Changed the "Deferred" dropdown to "Custom Shader". Then change the Shader option which appears to the shader you are using.

2017–06–08 Page published with no editorial review

Light Modes (**Shadowmask** and **Distance Shadowmask**) added in 5.6

Forward Rendering Path Details

Leave feedback

This page describes details of **Forward** rendering path.

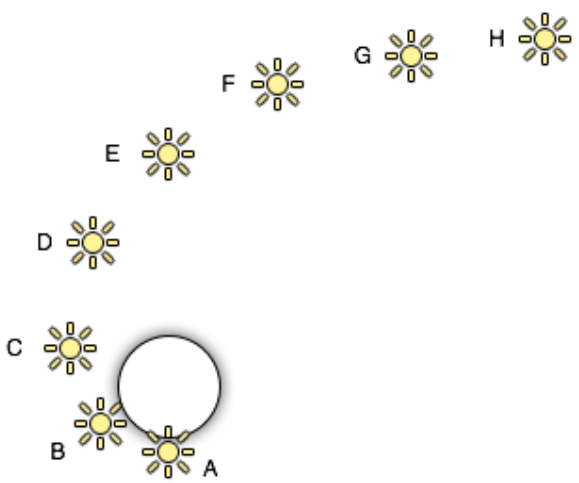
Forward **Rendering** path renders each object in one or more passes, depending on lights that affect the object. Lights themselves are also treated differently by **Forward Rendering**, depending on their settings and intensity.

Implementation Details

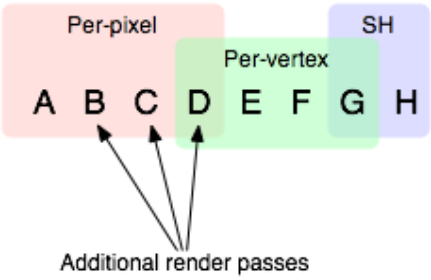
In Forward Rendering, some number of brightest lights that affect each object are rendered in fully per-pixel lit mode. Then, up to 4 point lights are calculated per-vertex. The other lights are computed as Spherical Harmonics (SH), which is much faster but is only an approximation. Whether a light will be a per-pixel light or not is dependent on this:

Lights that have their Render Mode set to **Not Important** are always per-vertex or SH.
Brightest directional light is always per-pixel.
Lights that have their Render Mode set to **Important** are always per-pixel.
If the above results in less lights than current **Pixel Light Count** Quality Setting, then more lights are rendered per-pixel, in order of decreasing brightness.
Rendering of each object happens as follows:

Base Pass applies one per-pixel directional light and all per-vertex/SH lights.
Other per-pixel lights are rendered in additional passes, one pass for each light.
For example, if there is some object that's affected by a number of lights (a circle in a picture below, affected by lights A to H):



Let's assume lights A to H have the same color and intensity and all of them have Auto **rendering mode**, so they would be sorted in exactly this order for this object. The brightest lights will be rendered in per-pixel lit mode (A to D), then up to 4 lights in per-vertex lit mode (D to G), and finally the rest of lights in SH (G to H):



Note that light groups overlap; for example last per-pixel light blends into per-vertex lit mode so there are less "light popping" as objects and lights move around.

Base Pass

Base pass renders object with one per-pixel directional light and all SH/vertex lights. This pass also adds any **lightmaps**, ambient and emissive lighting from the **shader**. Directional light rendered in this pass can have Shadows. Note that Lightmapped objects do not get illumination from SH lights.

Note that when "OnlyDirectional" pass flag is used in the shader, then the forward base pass only renders main directional light, ambient/lightprobe and lightmaps (SH and vertex lights are not included into pass data).

Additional Passes

Additional passes are rendered for each additional per-pixel light that affect this object. Lights in these passes by default do not have shadows (so in result, Forward Rendering supports one directional light with shadows), unless multi_compile_fwdadd_fullshadows variant shortcut is used.

Performance Considerations

Spherical Harmonics lights are *very* fast to render. They have a tiny cost on the CPU, and are *actually free* for the GPU to apply (that is, base pass always computes SH lighting; but due to the way SH lights work, the cost is exactly the same no matter how many SH lights are there).

The downsides of SH lights are:

They are computed at object's vertices, not **pixels**. This means they do not support light Cookies or normal maps.
SH lighting is very low frequency. You can't have sharp lighting transitions with SH lights. They are also only affecting the diffuse lighting (too low frequency for specular highlights).
SH lighting is not local; point or spot SH lights close to some surface will "look wrong".
In summary, SH lights are often good enough for small dynamic objects.

Legacy Deferred Lighting Rendering Path

Leave feedback

This page details the *Legacy Deferred Lighting (light prepass)* rendering path. See this article for a technical overview of deferred lighting.

Note: Deferred Lighting is considered a legacy feature starting with Unity 5.0, as it does not support some of the **rendering** features (e.g. Standard shader, reflection probes). New projects should consider using Deferred Shading rendering path instead.

NOTE: *Deferred rendering is not supported when using Orthographic projection. If the **camera**'s projection mode is set to Orthographic, the **camera** will always use Forward rendering.*

Overview

When using Deferred Lighting, there is no limit on the number of lights that can affect an object. All lights are evaluated per-pixel, which means that they all interact correctly with normal maps, **etc**. Additionally, all lights can have cookies and shadows.

Deferred lighting has the advantage that the processing overhead of lighting is proportional to the number of **pixels** the light shines on. This is determined by the size of the light volume in the **scene** regardless of how many objects it illuminates. Therefore, performance can be improved by keeping lights small. Deferred lighting also has highly consistent and predictable behaviour. The effect of each light is computed per-pixel, so there are no lighting computations that break down on large triangles.

On the downside, deferred lighting has no real support for anti-aliasing and can't handle semi-transparent objects (these will be rendered using forward rendering). There is also no support for the **Mesh** Renderer's Receive Shadows flag and **culling masks** are only supported in a limited way. You can only use up to four **culling masks**. That is, your culling **layer mask** must at least contain all layers minus four arbitrary layers, so 28 of the 32 layers must be set. Otherwise you will get graphical artefacts.

Requirements

It requires a graphics card with **Shader** Model 3.0 (or later), support for Depth render textures and two-sided stencil buffers. Most PC graphics cards made after 2004 support deferred lighting, including GeForce FX and later, Radeon X1300 and later, Intel 965 / GMA X3100 and later. On mobile, all OpenGL ES 3.0 capable GPUs support deferred lighting, and some of OpenGL ES 2.0 capable ones support it too (the ones that do support depth textures).

Performance Considerations

The rendering overhead of realtime lights in deferred lighting is proportional to the number of pixels illuminated by the light and is *not* dependent on scene complexity. So small point or spot lights are very cheap to render and if they are fully or partially occluded by scene objects then they are even cheaper.

Of course, lights with shadows are much more expensive than lights without shadows. In deferred lighting, shadow-casting objects still need to be rendered once or more for each shadow-casting light. Furthermore, the

lighting shader that applies shadows has a higher rendering overhead than the one used when shadows are disabled.

Implementation Details

When Deferred Lighting is used, the rendering process in Unity happens in three passes:

Base Pass: objects are rendered to produce screen-space buffers with depth, normals, and specular power.
Lighting pass: the previously generated buffers are used to compute lighting into another screen-space buffer.
Final pass: objects are rendered again. They fetch the computed lighting, combine it with color textures and add any ambient/emissive lighting.
Objects with shaders that can't handle deferred lighting are rendered after this process is complete, using the forward rendering path.

Base Pass

The base pass renders each object once. View space normals and specular power are rendered into a single ARGB32 Render Texture (with normals in RGB channels and specular power in A). If the platform and hardware allow the Z buffer to be read as a texture then depth is not explicitly rendered. If the Z buffer can't be accessed as a texture then depth is rendered in an additional rendering pass using shader replacement.

The result of the base pass is a Z buffer filled with the scene contents and a Render Texture with normals and specular power.

Lighting Pass

The lighting pass computes lighting based on depth, normals and specular power. Lighting is computed in screen space, so the time it takes to process is independent of scene complexity. The lighting buffer is a single ARGB32 Render Texture, with diffuse lighting in the RGB channels and monochrome specular lighting in the A channel. Lighting values are logarithmically encoded to provide greater dynamic range than is usually possible with an ARGB32 texture. When a camera has **HDR** rendering enabled, then lighting buffer is of ARGBHalf format, and logarithmic encoding is not performed.

Point and spot lights that do not cross the camera's near plane are rendered as (front faces of) 3D shapes, with the depth test against the scene enabled. Lights crossing the near plane are also rendered using 3D shapes, but as back faces with inverted depth test instead. This makes partially or fully occluded lights very cheap to render. If a light intersects both far and near camera planes at the same time, the above optimizations cannot be used, and the light is drawn as a tight **quad** with no depth testing.

The above doesn't apply to directional lights, which are always rendered as fullscreen quads.

If a light has shadows enabled then they are also rendered and applied in this pass. Note that shadows do not come for "free"; shadow casters need to be rendered and a more complex light shader must be applied.

The only lighting model available is Blinn-Phong. If a different model is wanted you can modify the lighting pass shader, by placing the modified version of the Internal-PrePassLighting.shader file from the Built-in shaders into a folder named "Resources" in your "Assets" folder. Then go to the Edit->Project Settings->Graphics window.

Changed the “Legacy Deferred” dropdown to “Custom Shader”. Then change the Shader option which appears to the lighting shader you are using.

Final Pass

The final pass produces the final rendered image. Here all objects are rendered again with shaders that fetch the lighting, combine it with textures and add any emissive lighting. **Lightmaps** are also applied in the final pass. Close to the camera, realtime lighting is used, and only baked indirect lighting is added. This crossfades into fully baked lighting further away from the camera.

Vertex Lit Rendering Path Details

Leave feedback

This page describes details of **Vertex Lit** rendering path.

Vertex Lit path generally renders each object in one pass, with lighting from all lights calculated for each vertex.

It's the fastest **rendering** path and has the widest hardware support.

Since all lighting is calculated at the vertex level, this rendering path does not support most per-pixel effects: shadows, normal mapping, light cookies, and highly detailed specular highlights are not supported.

Hardware Requirements for Unity's Graphics Features

Leave feedback

Summary

	Win/Mac/Linux	iOS/Android	Consoles
Deferred lighting	SM3.0, GPU support	-	Yes
Forward rendering	Yes	Yes	Yes
Vertex Lit rendering	Yes	Yes	-
Realtime Shadows	GPU support	GPU support	Yes
Image Effects	Yes	Yes	Yes
Programmable Shaders	Yes	Yes	Yes
Fixed Function Shaders	Yes	Yes	-

Realtime Shadows

Realtime Shadows work on most PC, console & mobile platforms. On Windows (Direct3D), the GPU also needs to support shadow mapping features; most discrete GPUs support that since 2003 and most integrated GPUs support that since 2007. Technically, on Direct3D 9 the GPU has to support D16/D24X8 or DF16/DF24 **texture formats**; and on OpenGL it has to support GL_ARB_depth_texture extension.

Mobile shadows (iOS/Android) require OpenGL ES 2.0 and GL_OES_depth_texture extension, or OpenGL ES 3.0. Most notably, the extension is **not** present on Tegra-based Android devices, so shadows do not work there.

Post-processing Effects

Post-processing effects require render-to-texture functionality, which is generally supported on anything made in this millenium.

Shaders

In Unity, you can write programmable or fixed function shaders. Programmable shaders are supported everywhere, and default to Shader Model 2.0 (desktop) and OpenGL ES 2.0 (mobile). It is possible to target higher shader models if you want more functionality. Fixed function is supported everywhere except consoles.

Graphics HOWTOs

Leave feedback

This section contains a list of common graphics-related tasks in Unity, and how to carry them out.

How do I Import Alpha Textures?

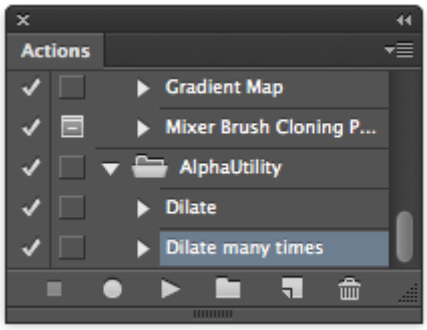
Leave feedback

Unity uses straight **alpha blending**. Hence, you need to expand the color layers... The alpha channel in Unity will be read from the first alpha channel in the Photoshop file.

Setting Up

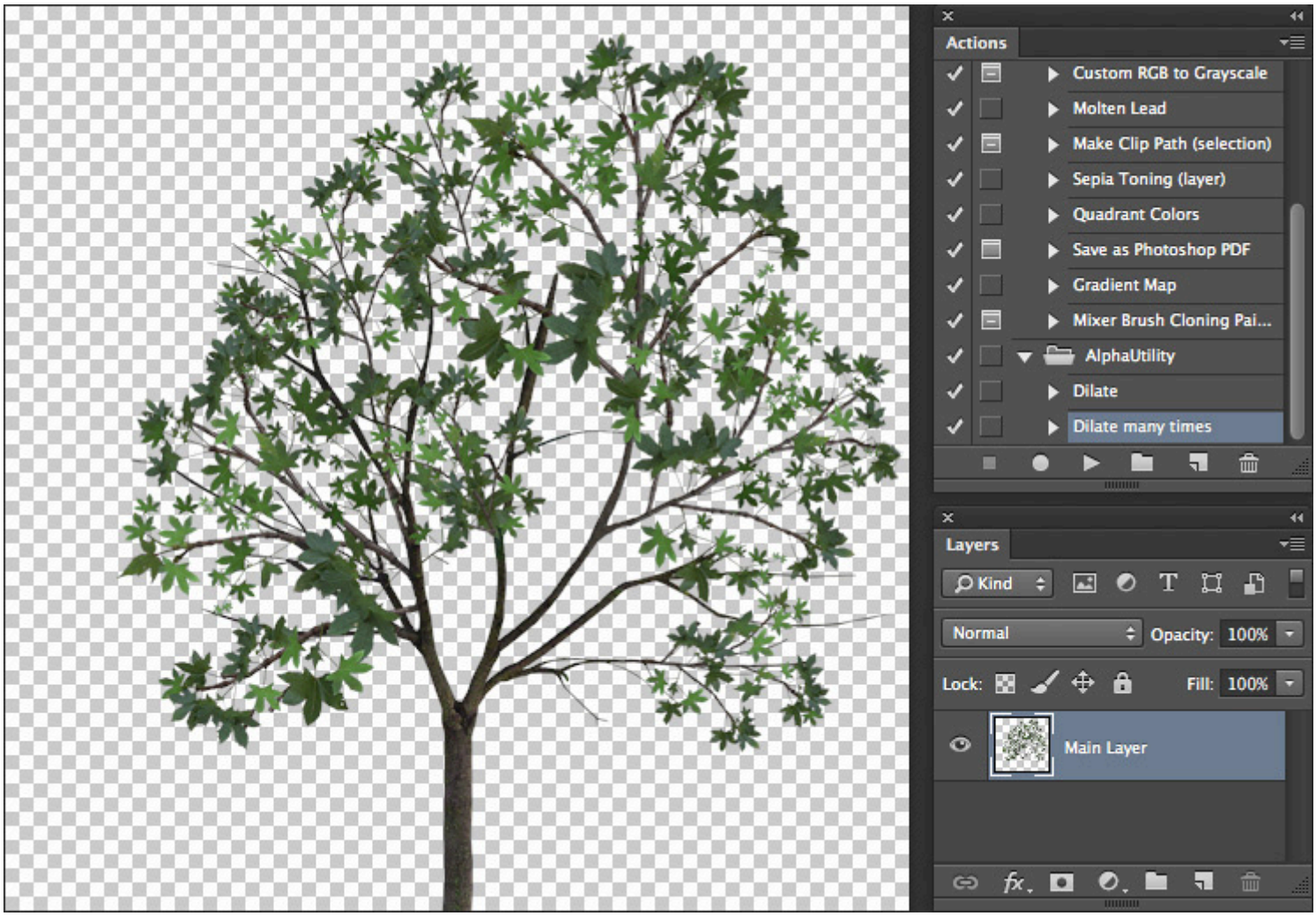
Before doing this, install these alpha utility photoshop actions: AlphaUtility.atn.zip

After installing, your Action Palette should contain a folder called AlphaUtility:



Getting Alpha Right

Let's assume you have your alpha texture on a transparent layer inside photoshop. Something like this:

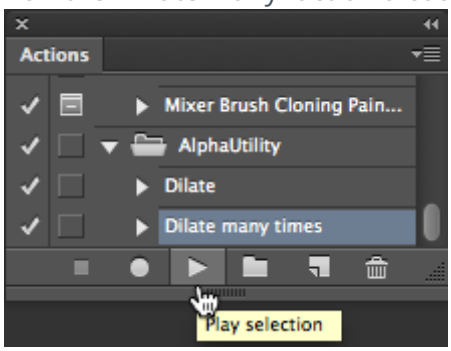


Duplicate the layer

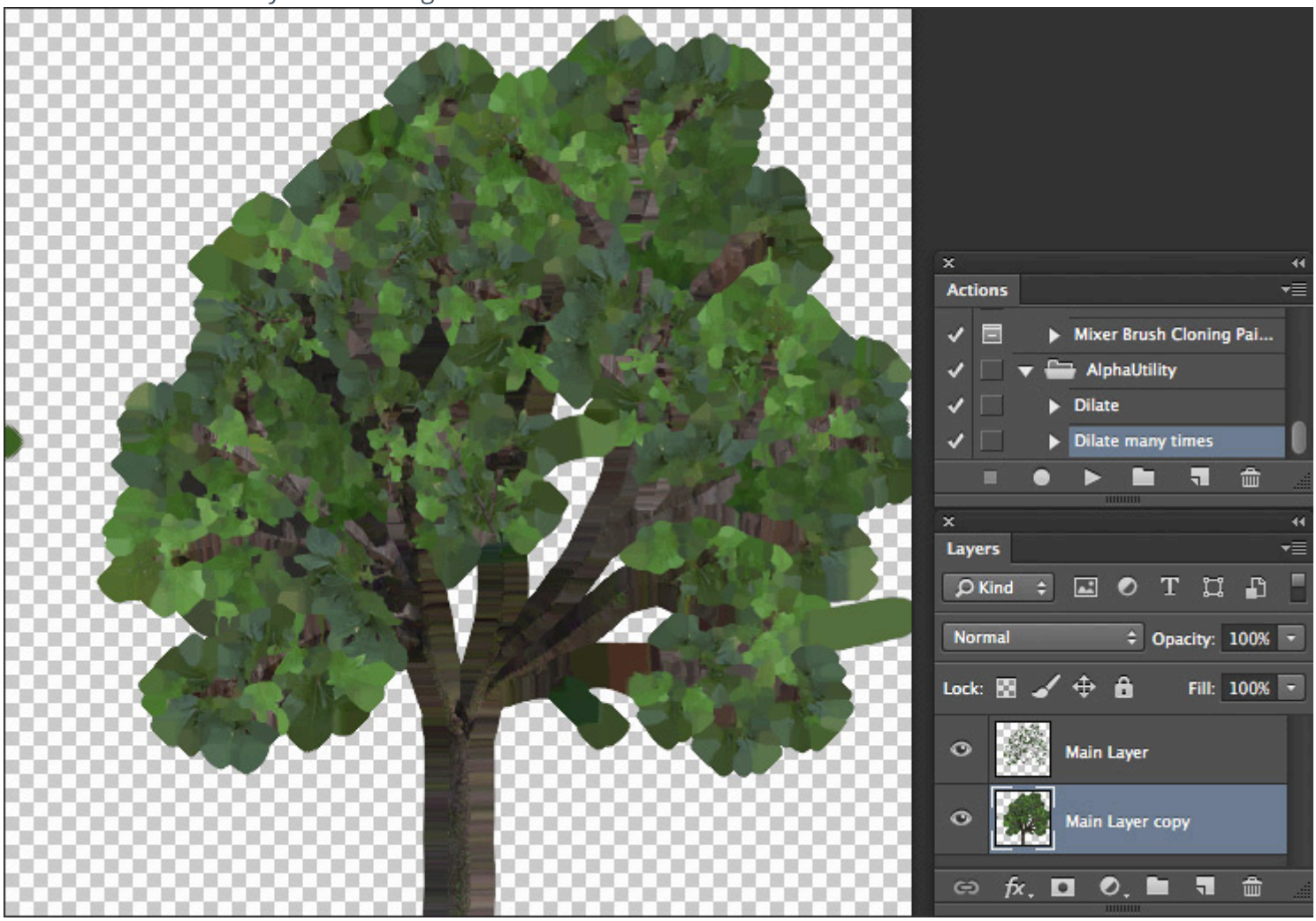
Select the lowest layer. This will be source for the dilation of the background.

Select **Layer->Matting->Defringe** and apply with the default properties

Run the “Dilate Many” action a couple of times. This will expand the background into a new layer.



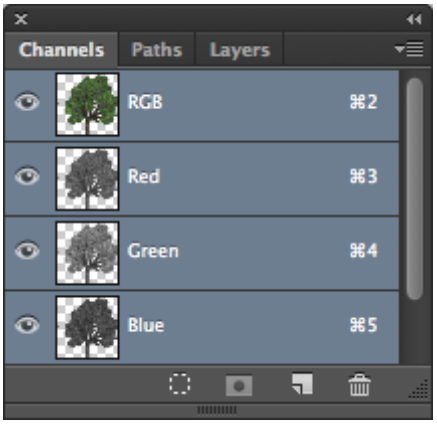
Select all the dilation layers and merge them with **Command-E**



Create a solid color layer at the bottom of your image stack. This should match the general color of your document (in this case, greenish). Note that without this layer Unity will take alpha from merged transparency of all layers.

Now we need to copy the transparency into the alpha layer.

Set the selection to be the contents of your main layer by Command-clicking on it in the Layer Palette.
Switch to the channels palette.
Create a new channel from the transparency.



Save your PSD file - you are now ready to go.

Extra

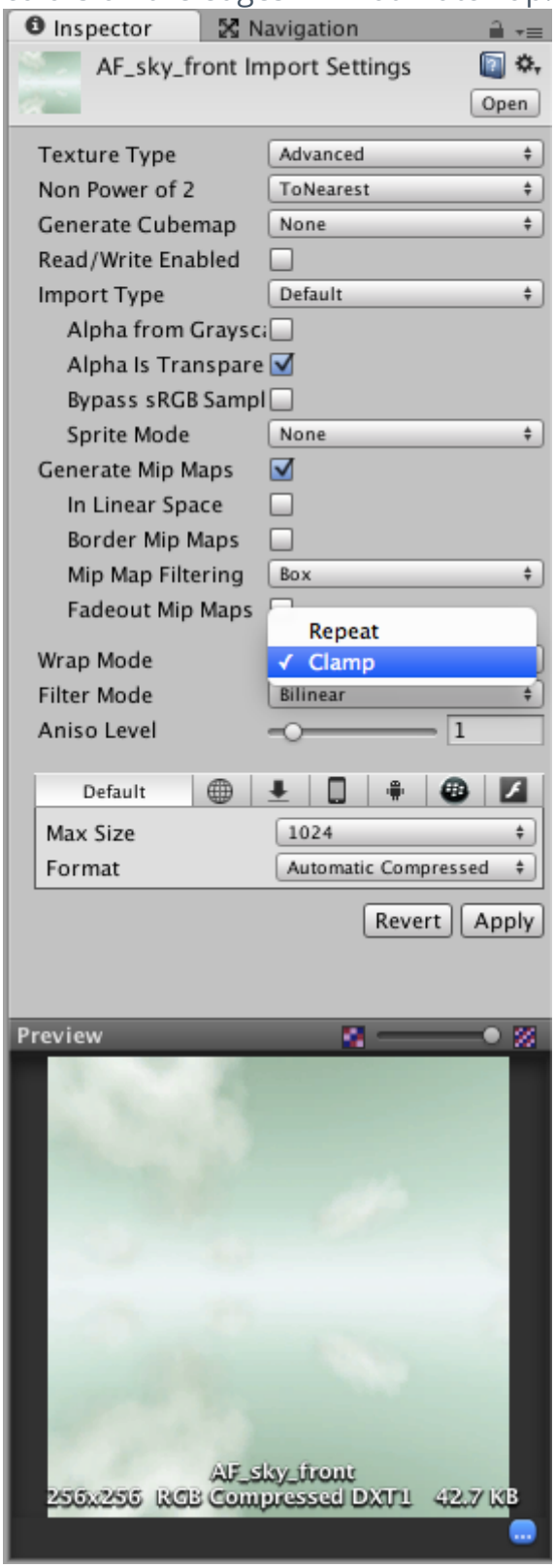
Note that if your image contains transparency (after merging layers), then Unity will take alpha from merged transparency of all layers and it will ignore Alpha masks. A workaround for that is to create a layer with solid color as described in item 6 under “Getting Alpha Right”

How do I Make a Skybox?

Leave feedback

A **Skybox** is a 6-sided cube that is drawn behind all graphics in the game. Here are the steps to create one:

Make 6 textures that correspond to each of the 6 sides of the skybox and put them into your project's **Assets** folder.
For each texture you need to change the wrap mode from **Repeat** to **Clamp**. If you don't do this colors on the edges will not match up:

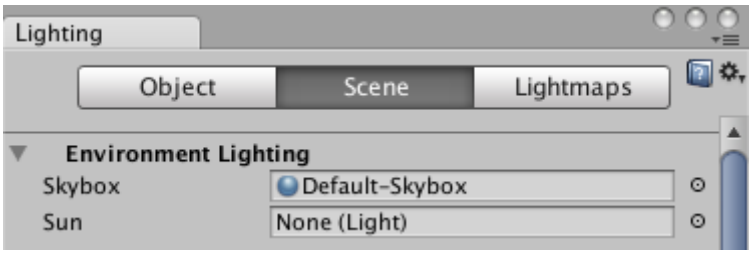


Create a new **Material** by choosing **Assets->Create->Material** from the menu bar.
Select the **shader** drop-down in the top of the **Inspector**, choose **Skybox/6 Sided**.
Assign the 6 textures to each texture slot in the material. You can do this by dragging each texture from the **Project View** onto the corresponding slots.

In this screen shot the textures have been taken from the 4.x StandardAssets/Skyboxes/Textures folder. Note that these textures are already used in SkyBoxes.

To Assign the skybox to the **scene** you're working on:

Choose **Window** > **Rendering** > **Lighting Settomgs** from the menu bar.
In the window that appears select the Scene tab.
Drag the new Skybox Material to the **Skybox** slot.


Lighting
Object
Scene
Lightmaps
Environment Lighting
Skybox
Default-Skybox
Sun
None (Light)

How do I make a Mesh Particle Emitter? (Legacy Particle System)

Leave feedback

Mesh Particle Emitters are generally used when you need high control over where to emit particles.

For example, when you want to create a flaming sword:

Drag a **mesh** into the **scene**.
Remove the **Mesh Renderer** by right-clicking on the **Mesh Renderer's Inspector** title bar and choose **Remove Component**.
Choose **Mesh Particle Emitter** from the **Component->Effects->Legacy Particles** menu.
Choose **Particle Animator** from the **Component->Effects->Legacy Particles** menu.
Choose **Particle Renderer** from the **Component->Effects->Legacy Particles** menu.
You should now see particles emitting from the mesh.

Play around with the values in the Mesh Particle Emitter.

Especially enable **Interpolate Triangles** in the Mesh Particle Emitter Inspector and set **Min Normal Velocity** and **Max Normal Velocity** to 1.

To customize the look of the particles that are emitted:

Choose **Assets->Create->Material** from the menu bar.
In the Material Inspector, select **Particles->Additive** from the shader drop-down.
Drag & drop a texture from the **Project view** onto the texture slot in the Material Inspector.
Drag the Material from the Project View onto the **Particle System** in the **Scene View**.
You should now see textured particles emitting from the mesh.

See Also

Mesh Particle Emitter Component Reference page

How do I make a Spot Light Cookie?

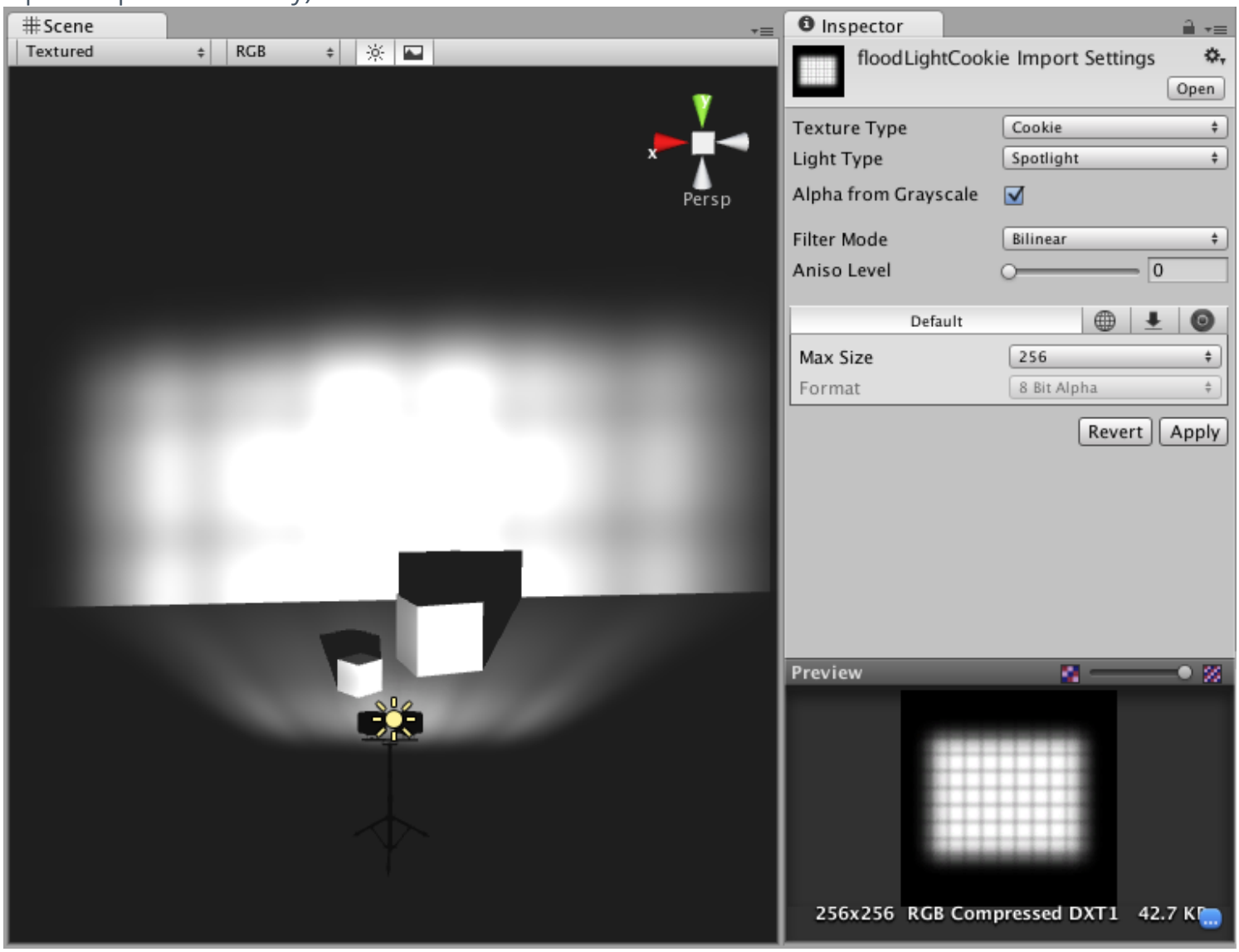
Leave feedback

Unity ships with a few **Light Cookies** in the Standard Assets. When the Standard Assets are imported to your project, they can be found in **Standard Assets->Light Cookies**. This page shows you how to create your own.

A great way to add a lot of visual detail to your **scenes** is to use cookies - grayscale textures you use to control the precise look of in-game lighting. This is fantastic for making moving clouds and giving an impression of dense foliage. The Light Component Reference page has more info on all this, but the main thing is that for textures to be usable for cookies, the following properties need to be set:

To create a light cookie for a spot light:

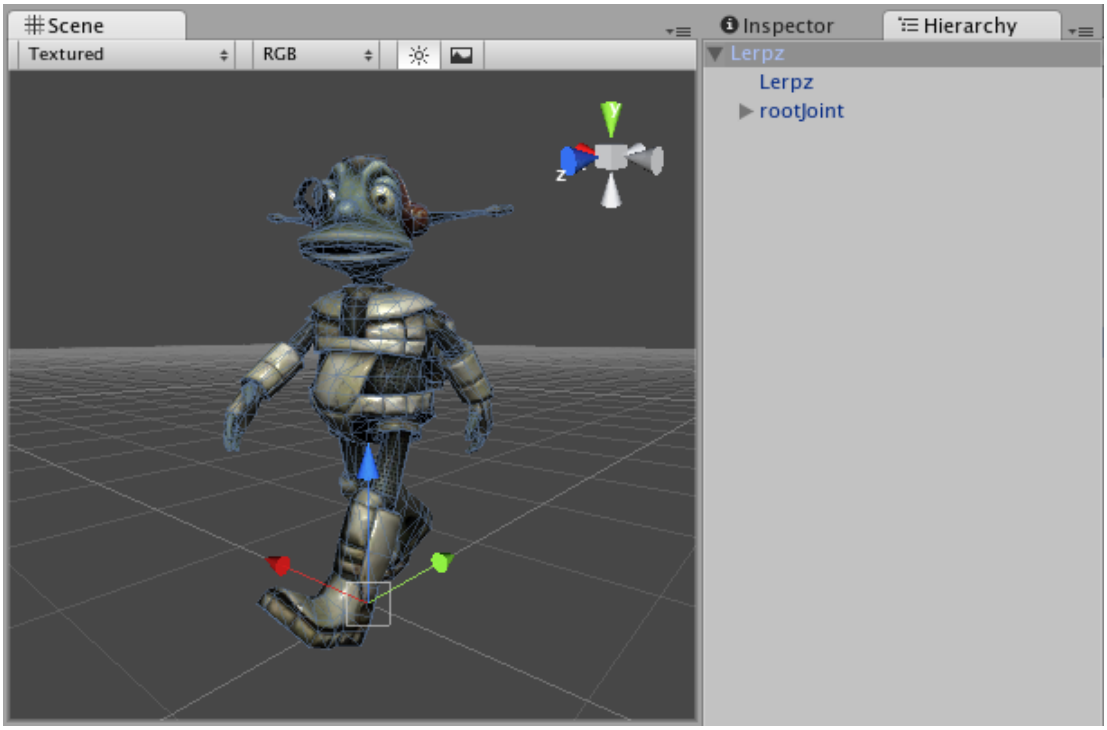
Paint a cookie texture in Photoshop. The image should be greyscale. White **pixels** means full lighting intensity, black **pixels** mean no lighting. The borders of the texture need to be completely black, otherwise the light will appear to leak outside of the spotlight.
In the **Texture Inspector** change the **Texture Type** to **Cookie**
Enable **Alpha From Grayscale** (this way you can make a grayscale cookie and Unity converts it to an alpha map automatically)



How do I fix the rotation of an imported model?

Leave feedback

Some 3D art packages export their models so that the z-axis faces upward. Most of the standard **scripts** in Unity assume that the y-axis represents **up** in your 3D world. It is usually easier to fix the rotation in Unity than to modify the **scripts** to make things fit.



Your model with z-axis points upwards

If at all possible it is recommended that you fix the model in your 3D modelling application to have the y-axis face upwards before exporting.

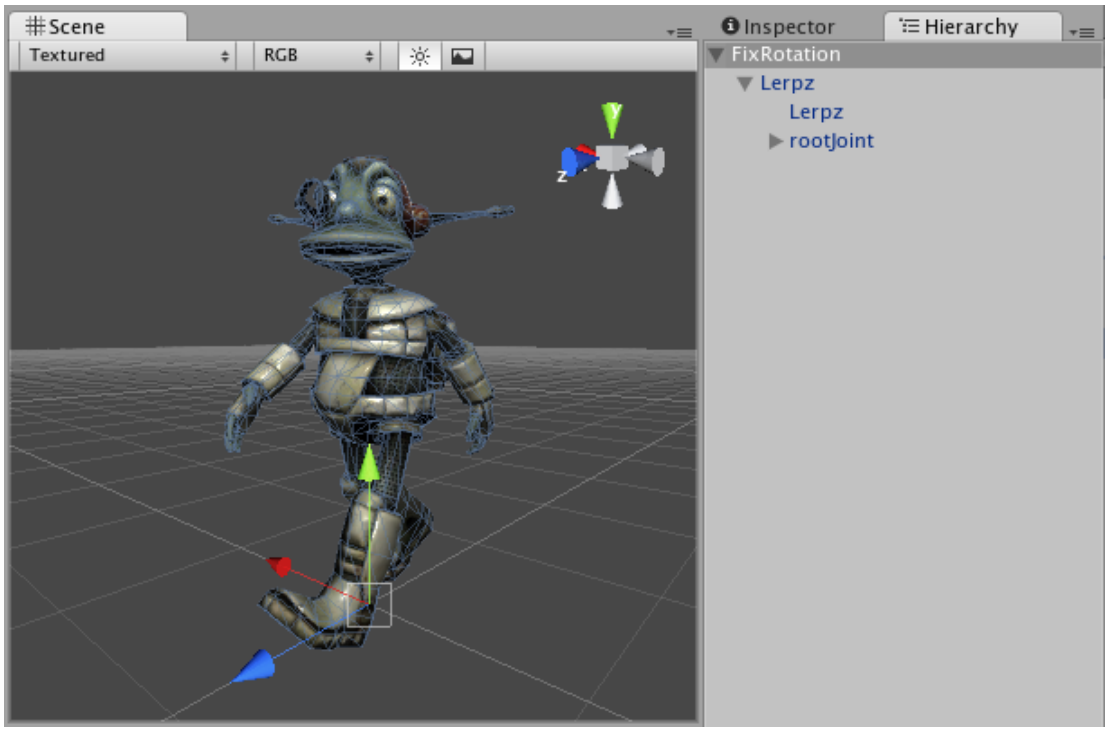
If this is not possible, you can fix it in Unity by adding an extra parent transform:

Create an empty **GameObject** using the **GameObject->Create Empty** menu

Position the new GameObject so that it is at the center of your **mesh** or whichever point you want your object to rotate around.

Drag the mesh onto the empty GameObject

You have now made your mesh a **Child** of an empty GameObject with the correct orientation. Whenever writing scripts that make use of the y-axis as up, attach them to the **Parent** empty GameObject.



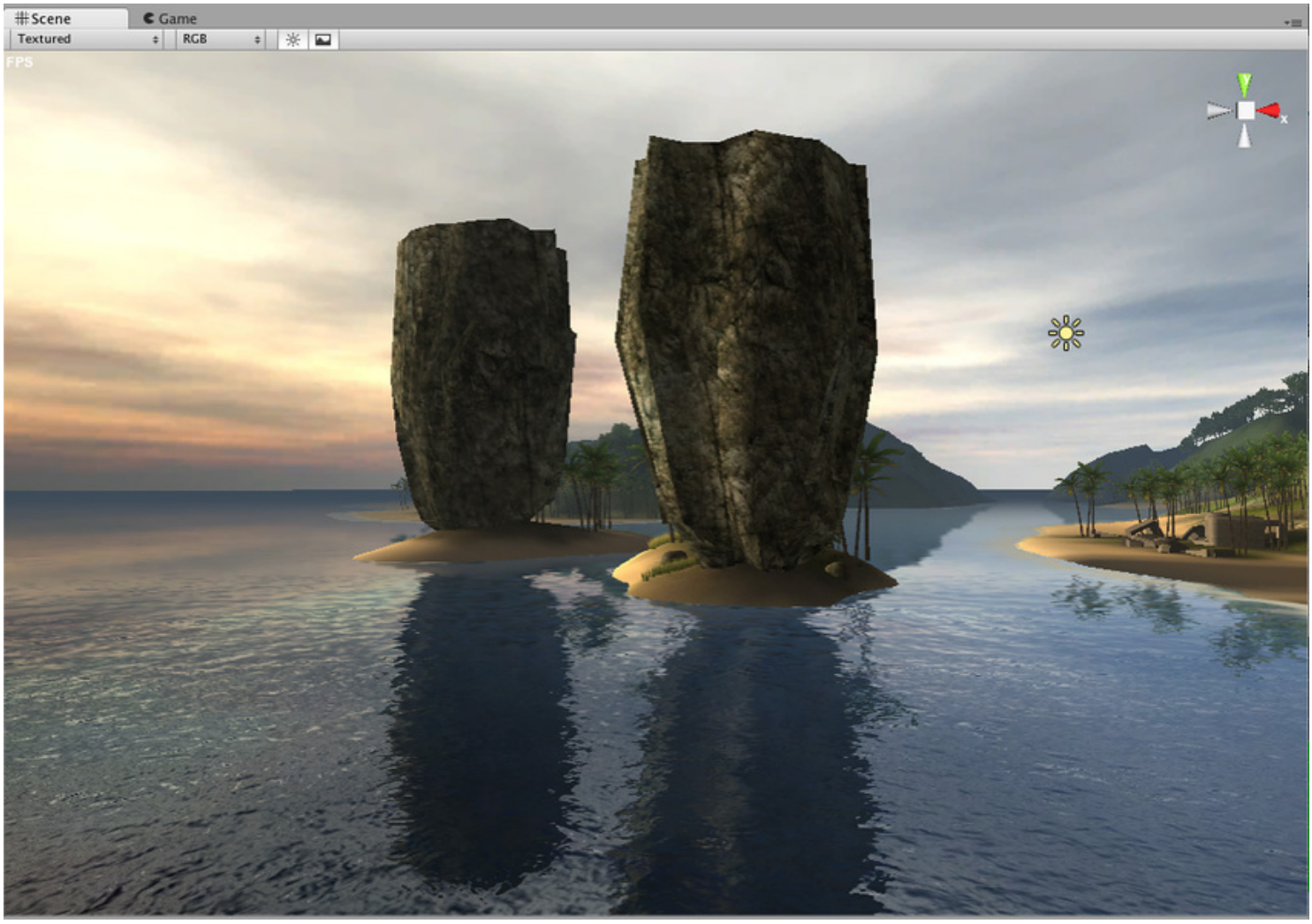
The model with an extra empty transform

Water in Unity

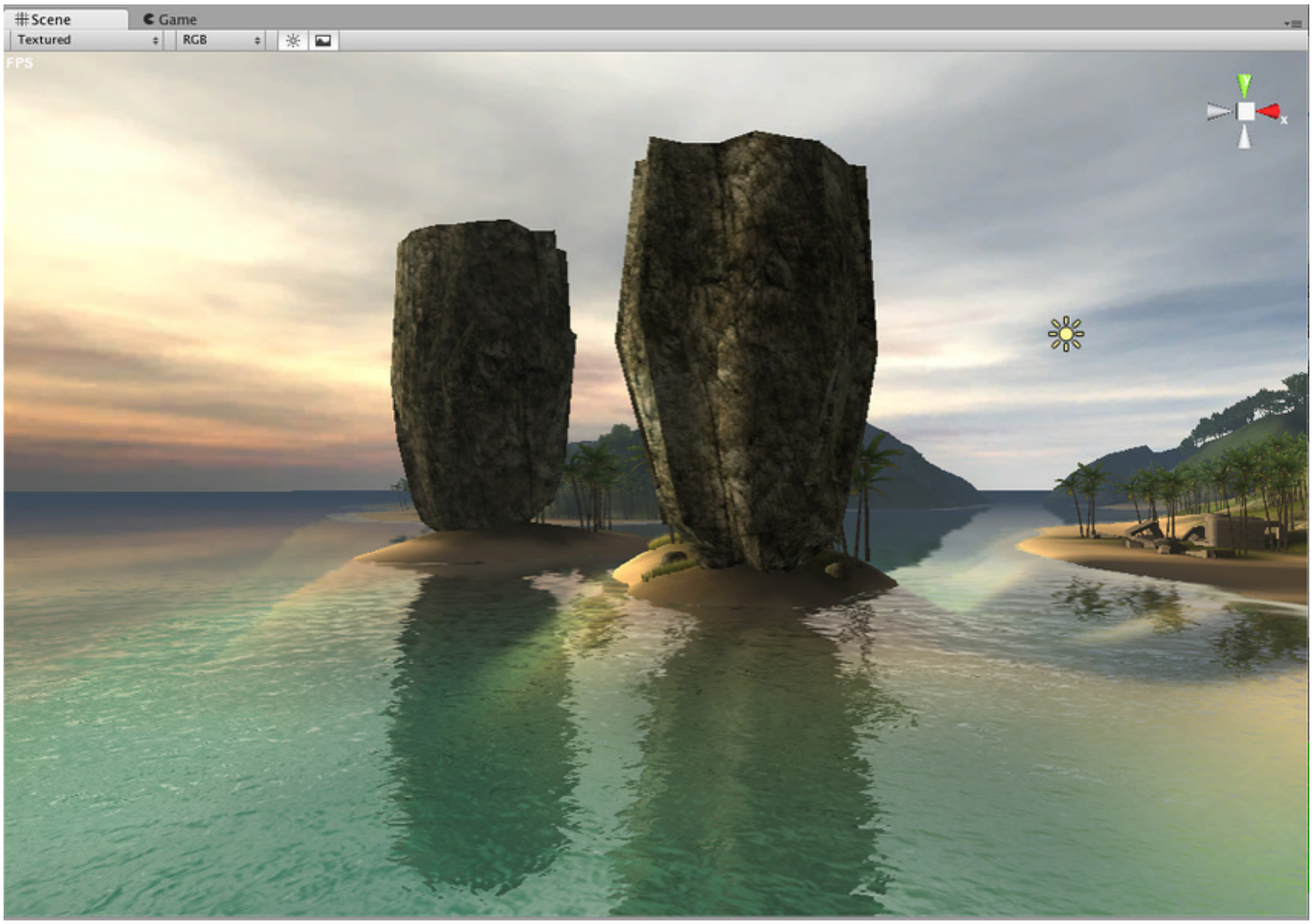
Leave feedback

Unity includes several water **Prefabs** (including the necessary Shaders, scripts and art Assets) within the Standard Assets packages. Separate daylight and nighttime water Prefabs are provided.

Note that the water reflections described in this document do not work in VR.



Reflective daylight water

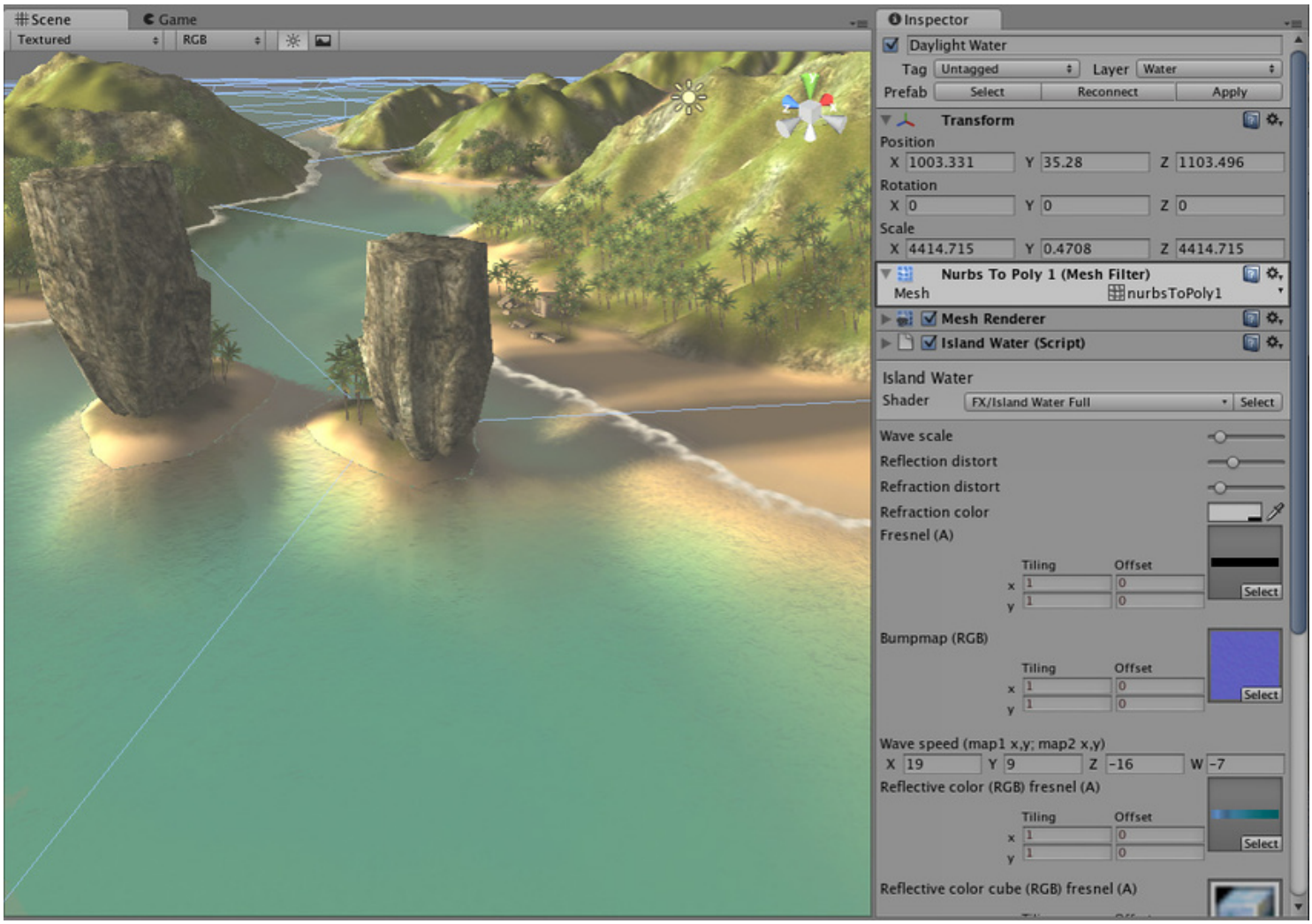


Reflective/Refractive daylight water

Setting up water

Place one of the existing water Prefabs into your **Scene**. Make sure you have the Standard Assets installed:

Simple **Daylight Simple Water** and **Nighttime Simple Water** in **Standard Assets** > **Water**.
Fancier water - **Daylight Water** and **Nighttime Water** in **Pro Standard Assets** > **Water** (this needs some Assets from **Standard Assets** > **Water** as well). Water mode (Simple, Reflective, Refractive) can be set in the Inspector.
The Prefab uses an oval-shaped Mesh for the water. If you need to use a different Mesh, change it in the **Mesh Filter** of the water **GameObject**:



Creating water from scratch (Advanced)

Simple water requires attaching a script to a plane-like mesh and using the water **shader**:

Have mesh for the water. This should be a flat mesh, oriented horizontally. UV coordinates are not required. The water GameObject should use the Water **Layer**, which you can set in the **Inspector**.
Attach the **WaterSimple** script (from **Standard Assets/Water/Sources**) to the GameObject.
Use the **FX/Water (simple)** Shader in the Material, or tweak one of the provided water Materials (**Daylight Simple Water** or **Nighttime Simple Water**).
Reflective/refractive water requires similar steps to set up from scratch:

Create a Mesh for the water. This should be a flat Mesh, oriented horizontally. UV coordinates are not required. The water GameObject should use the water **Layer**, which you can set in the Inspector.
Attach the **Water** script (from **Pro Standard Assets/Water/Sources**) to the GameObject (Water rendering mode can be set in the Inspector: **Simple**, **Reflective** or **Refractive**.)
Use the **FX/Water** Shader in the Material, or tweak one of the provided water Materials (**Daylight Water** or **Nighttime Water**).

Properties in water Materials

These properties are used in the **Reflective** and **Refractive** water Shaders. Most of them are used in the **Simple** water Shader as well.

Property:	*Function:*
Wave scale	Scaling of waves normal map. The smaller the value, the larger water waves.
Reflection/refraction distort	How much reflection/refraction is distorted by the waves normal map.

Property:	*Function:*
Refraction color	Additional tint for refraction.
Environment reflection/refraction	**Render textures** for real-time reflection and refraction.
Normalmap	Defines the shape of the waves. The final waves are produced by combining these two normal maps, each scrolling at different direction, scale and speed. The second normal map is half as large as the first one.
Wave speed	Scrolling speed for first normal map (1st and 2nd numbers) and the second normal map (3rd and 4th numbers).
Fresnel	A texture with alpha channel controlling the **Fresnel effect** - how much reflection vs. refraction is visible, based on viewing angle.

The rest of the properties are not used by the **Reflective** and **Refractive** Shaders, but need to be set up in case the user's video card does not support it and must fallback to the simpler shader:

Property:	*Function:*
Reflective color/cube and fresnel	A texture that defines water color (RGB) and Fresnel effect (A) based on viewing angle.
Horizon color	The color of the water at horizon. `this is only used in the **Simple** water Shader.
Fallback texture	Define the fallback Texture used to represent the water on really old video cards, if none of the better looking Shaders can run.

Art Asset best practice guide

Leave feedback

Unity supports textured 3D models from a variety of programs or sources. This short guide has been put together by games artists with developers at Unity to help you create **Assets** that work better and more efficiently in your Unity project.

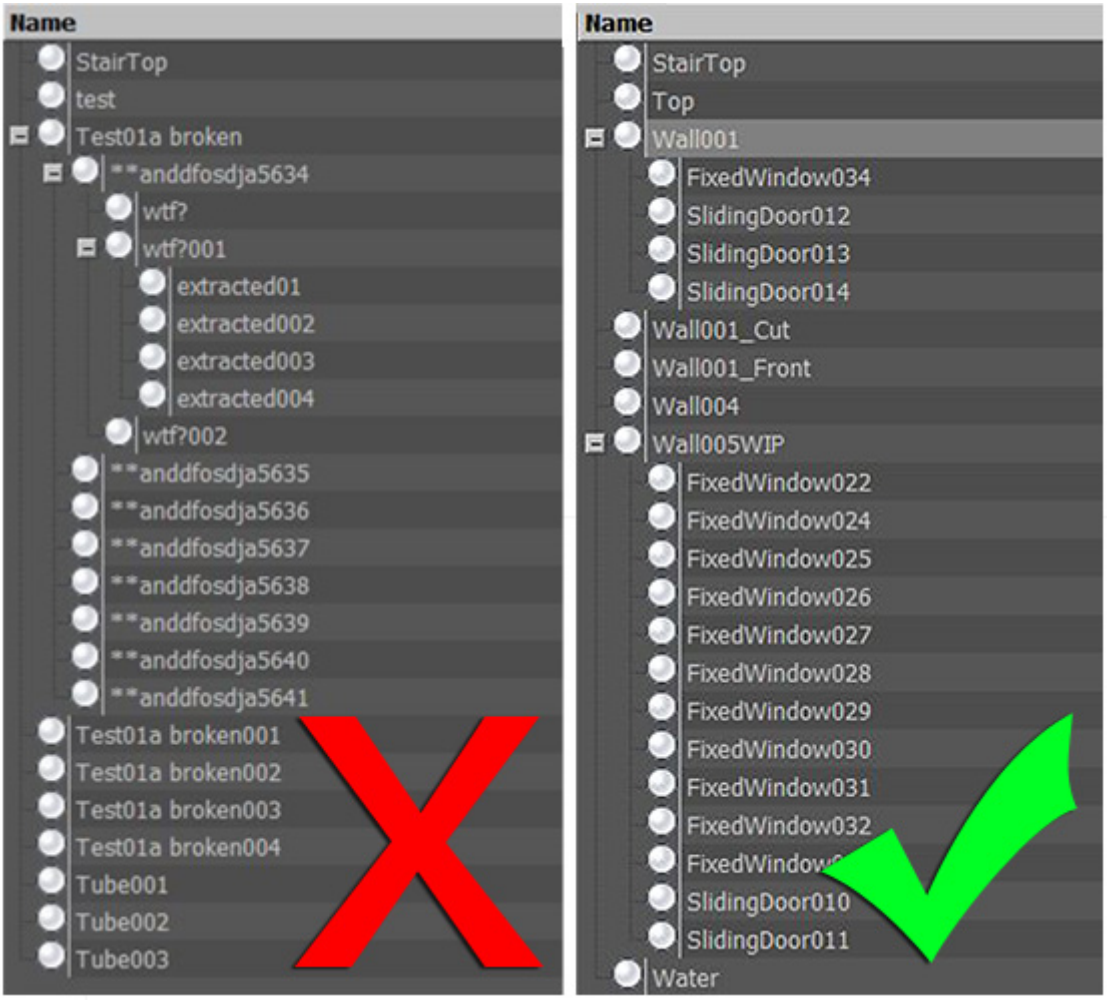
Scale and units

Working to scale can be important for both lighting and physics simulation.

Set your system and project units to Metric for your software to work consistently with Unity.
Be aware that different systems use different units - for example, the system unit default for Max is inches while in Maya it is centimeters.
Unity has different scaling for importing FBX vs. importing native 3D modeling software files. Make sure you check the FBX import scale settings. For example, if you want to achieve **Scale Factor** = 1 and **Object Transform Scale** = 1, use one of the proprietary file formats and set the **Convert Units** option.
If in doubt, export a meter cube with your **Scene** to match in Unity.
Animation frame rate defaults can be different in different packages. Because of this, it is a good idea to set this consistently across your pipeline (for example, at 30fps).

Files and objects

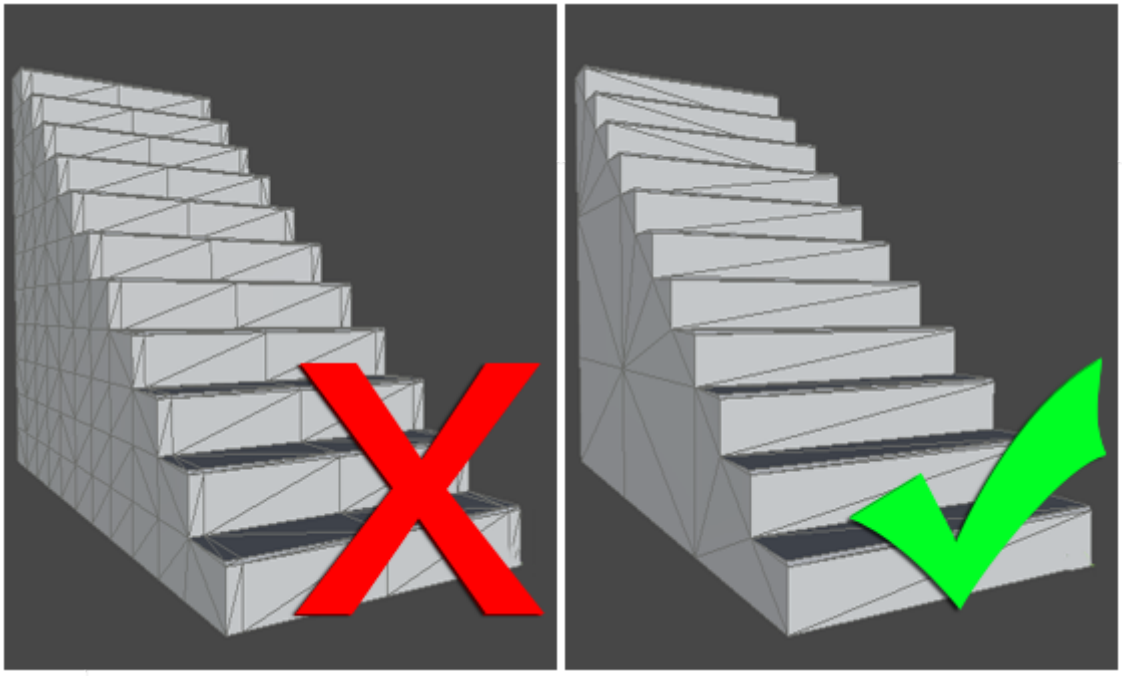
Name objects in your Scene sensibly and uniquely. This helps you locate and troubleshoot specific Meshes in your project.
Avoid special characters such as `*()?"#$`.
Use simple but descriptive names for both objects and files in order to allow for duplication later.
Keep your hierarchies as simple as you can.
With big projects in your 3D application, consider having a working file outside your Unity project directory. This can often save time when running updates and importing unnecessary data.



Sensibly named objects help you find things quickly

Mesh

Build with an efficient topology. Use polygons only where you need them.
Optimise your geometry if it has too many polygons. Many character models need to be intelligently optimized or even rebuilt by an artist, especially if sourced or built from:
3D capture data
Poser
Zbrush
Other high density NURBS patch models designed for render
Where you can afford them, evenly spaced polygons in buildings, landscape and architecture will help spread lighting and avoid awkward kinks.
Avoid really long thin triangles.



Stairway to framerate heaven

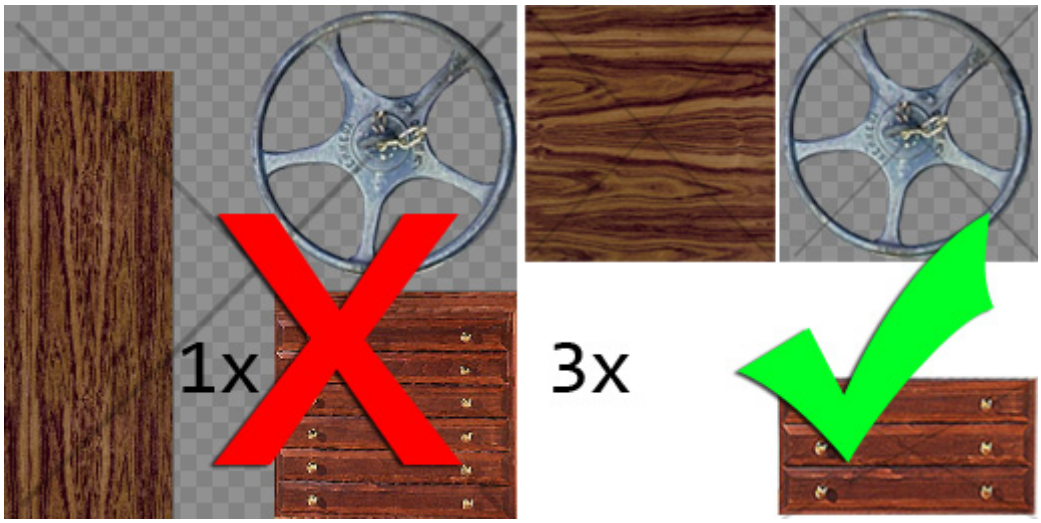
The method you use to construct objects can have a massive effect on the number of polygons, especially when not optimized. In this diagram, the same shape **mesh** has 156 triangles on the right and 726 on the left. 726 may not sound like a great deal of polygons, but if this is used 40 times in a level, you will really start to see the savings. A good rule of thumb is often to start simple and add detail where needed. It's always easier to add polygons than take them away.

Textures

If you author your textures to a power of two (for example, 512x512 or 256x1024), the textures will be more efficient and won't need rescaling at build time. You can use up to 4096x4096 **pixels**, but 2048x2048 is the highest available on many graphics cards and platforms.

Search online for expert advice on creating good textures, but some of these guidelines can help you get the most efficient results from your project:

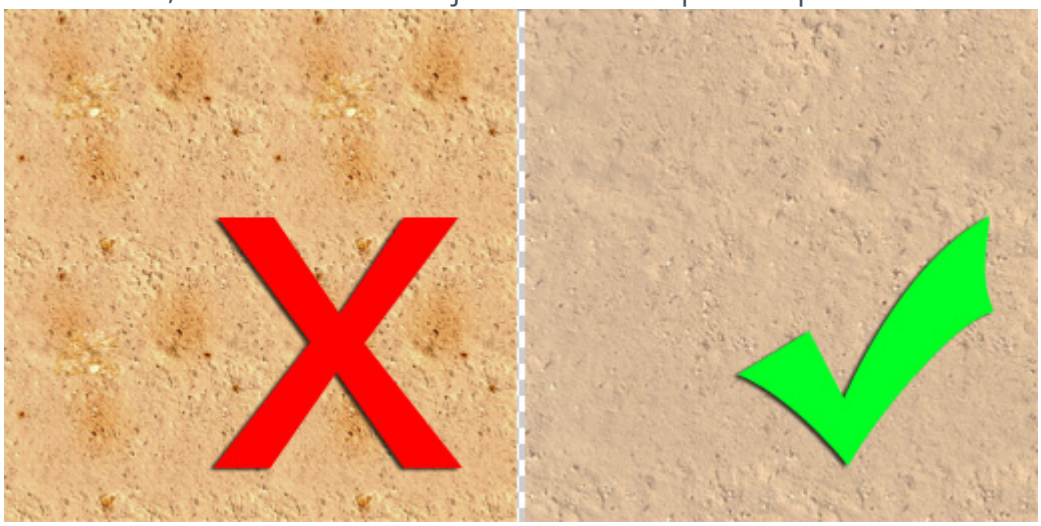
Work with a high-resolution source file outside your Unity project (such as a .psd or Gimp file). You can always downsize from the source, but not the other way round.
Use the Texture resolution output you require in your Scene (save a copy, for example a 256x256 optimised .png or a .tga file). You can make a judgement based on where the Texture will be seen and where it is mapped.
Store your output Texture files together in your Unity project (for example, in \Assets\textures).
Make sure your 3D working file refers to the same Textures, for consistency when you save or export.
Make use of the available space in your Texture, but be aware of different Materials requiring different parts of the same Texture. You can end up using or loading that Texture multiple times.
For alpha and elements that may require different **Shaders**, separate the Textures. For example, the single Texture, below left, has been replaced by three smaller Textures, below right.



One Texture (left) versus three Textures (right)

Make use of tiling Textures which seamlessly repeat. This allows you to use better resolution repeating over space.

Remove easily noticeable repeating elements from your bitmap, and be careful with contrast. To add details, use decals and objects to break up the repeats.



Tiling Textures

Unity takes care of **compression** for the output platform, so unless your source is already a .jpg of the correct resolution, it's better to use a lossless format for your Textures.

When creating a Texture page from photographs, reduce the page to individual modular sections that can repeat. For example, you don't need 12 of the same window using up Texture space. This means you can have more pixel detail for that one window.

Less is more when it comes to windows

Materials

Organize and name the materials in your Scene. This way you can find and edit your materials in Unity more easily when they've imported.
You can choose to create Materials in Unity from either:
<modelname> - <material name> or
<texture name> Make sure you know which one you want.
Settings for Materials in your native package will not all be imported to Unity:
Diffuse Colour, **Diffuse Texture** and **Names** are usually supported.
Shader model, specular, normal, other secondary Textures and substance material settings are not recognised or imported.

Import and export

Unity can use two types of files: saved 3D application files, and exported 3D formats. Which you decide to use can be quite important. For more information, see Model file formats.

2018–04–25 Page amended with limited editorial review

2018–09–26 Page amended with limited editorial review

Importing models from 3D modeling software

Leave feedback

There are two ways to import Models into Unity:

Drag the **Model file** from your file browser straight into the Unity Project window.
Copy the Model file into the Project's Assets folder.
Select the file in the **Project** view and navigate to the **Model** tab in the **Inspector** window to configure import options. See the [Model Import Settings window] reference documentation for details.

Note: You must store Textures in a folder called **Textures**, placed inside the **Assets** folder (next to the exported Mesh) within your Unity Project. This enables the Unity Editor to find the Textures and connect them to the generated Materials. For more information, see the Importing Textures documentation.

See also

Modeling optimized characters
Mesh import settings
Mesh components
Fixing the rotation of an imported Model

2018–04–25 Page amended with limited editorial review

How to do Stereoscopic Rendering

Leave feedback

Stereoscopic rendering for DirectX11.1's stereoscopic 3d support.

The minimum requirements are:

Windows 10
Graphic card that supports DirectX 11.
The graphics card driver needs to be set up with stereo support, and you need to use a dual DVI or DisplayPort cable; a single DVI is not enough.
The Stereoscopic checkbox in the **Player Settings** is strictly for DirectX 11.1's stereoscopic 3d support. It doesn't currently use AMD's **quad** buffer extension. Make sure that this sample works on your machine. Stereo support works both in fullscreen and windowed mode.

When you launch the game, hold shift to bring up the resolution dialog. There will be a checkbox in the resolution dialog for Stereo3D if a capable display is detected. Regarding the API, there are a few options on **Camera**: stereoEnabled, stereoSeparation, stereoConvergence. Use these to tweak the effect. You will need only one **camera** in the **scene**, the rendering of the two eyes is handled by those parameters.

Note that this checkbox is not for VR headsets.

Check your setup using this sample.
Check the Stereoscopic Rendering and Use Direct3D 11 checkboxes in the Player Settings.
Publish it as 32-bit and 64-bit applications.
Try it with single camera and double cameras.
Hold Shift when launching the application to see Stereo 3D checkbox in the resolution dialog. The resolution dialog might be suppressed or always enabled depending on the project's player settings.
Note: Currently, setting Unity to render in linear color space breaks stereoscopic rendering. This appears to be a Direct3D limitation. It also appears that the `camera.stereoconvergence` param has no effect at all if you have some realtime shadows enabled (in forward rendering). In Deferred Lighting, you will get some shadows, but insconsistent between left & right eye.

Graphics Tutorials

Leave feedback

This section contains a number of graphics tutorials to help you learn Unity's graphics systems.

For a more comprehensive set of lessons and tutorials, see the Graphics section on the Unity Learn page.

Shaders: ShaderLab and fixed function shaders

Leave feedback

This tutorial teaches you the first steps of creating your own shaders, to help you control the look of your game and optimise the performance of the graphics.

Unity is equipped with a powerful shading and material language called **ShaderLab**. In style it is similar to CgFX and Direct3D Effects (.FX) languages - it describes everything needed to display a Material.

Shaders describe properties that are exposed in Unity's Material Inspector and multiple shader implementations (**SubShaders**) targeted at different graphics hardware capabilities, each describing complete graphics hardware **rendering** state, and vertex/fragment programs to use. Shader programs are written in the high-level Cg/HLSL programming language.

In this tutorial we'll describe how to write *very simple* shaders using so-called "fixed function" notation. In the next chapter we'll introduce vertex and fragment shader programs. We assume that the reader has a basic understanding of OpenGL or Direct3D render states, and has some knowledge of HLSL, Cg, GLSL or Metal shader programming languages.

Getting started

To create a new shader, either choose **Assets > Create > Shader > Unlit Shader** from the main menu, or duplicate an existing shader and work from that. The new shader can be edited by double-clicking it in the Project View.

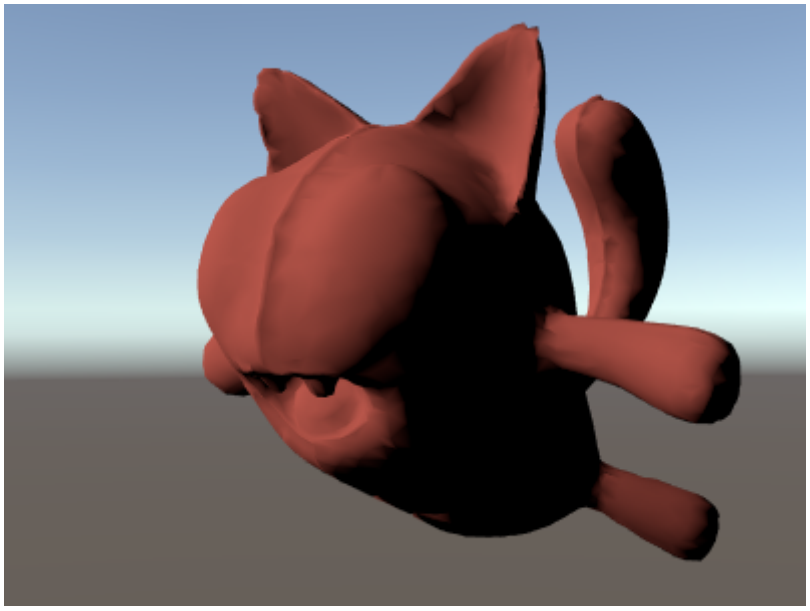
Unity has a way of writing *very simple* shaders in so-called "fixed-function" notation. We'll start with this for simplicity. Internally the fixed function shaders are converted to regular vertex and fragment programs at shader import time.

We'll start with a very basic shader:

```
Shader "Tutorial/Basic" {
    Properties {
        _Color ("Main Color", Color) = (1,0.5,0.5,1)
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

This simple shader demonstrates one of the most basic shaders possible. It defines a color property called **Main Color** and assigns it a default pink color (red=100% green=50% blue=50% alpha=100%). It then renders the object by invoking a **Pass** and in that pass setting the diffuse material **component** to the property **_Color** and turning on per-vertex lighting.

To test this shader, create a new material, select the shader from the drop-down menu (**Tutorial->Basic**) and assign the Material to some object. Tweak the color in the Material Inspector and watch the changes. Time to move onto more complex things!



Basic vertex lighting

If you open an existing complex shader, it can be a bit hard to get a good overview. To get you started, we will dissect the built-in **VertexLit** shader that ships with Unity. This shader uses fixed-function pipeline to do standard per-vertex lighting.

```
Shader "VertexLit" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0.5)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmisive Color", Color) = (0,0,0,0)
        _Shininess ("Shininess", Range (0.01, 1)) = 0.7
        _MainTex ("Base (RGB)", 2D) = "white" { }
    }

    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
                Ambient [_Color]
                Shininess [_Shininess]
                Specular [_SpecColor]
                Emission [_Emission]
```

```
            }
            Lighting On
            SeparateSpecular On
            SetTexture [_MainTex] {
                constantColor [_Color]
                Combine texture * primary DOUBLE, texture * constant
            }
        }
    }
}
```

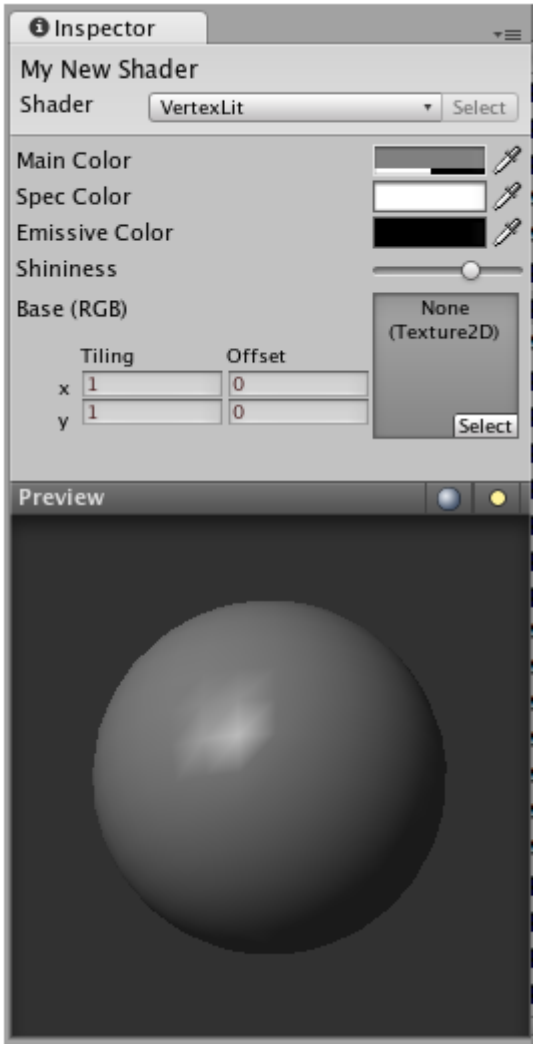
All shaders start with the keyword Shader followed by a string that represents the name of the shader. This is the name that is shown in the **Inspector**. All code for this shader must be put within the curly braces after it: **{ }** (called a block).

The name should be short and descriptive. It does not have to match the **.shader** file name.
To put shaders in submenus in Unity, use slashes - e.g. **MyShaders/Test** would be shown as **Test** in a submenu called **MyShaders**, or **MyShaders->Test**.
The shader is composed of a **Properties** block followed by **SubShader** blocks. Each of these is described in sections below.

Properties

At the beginning of the shader block you can define any properties that artists can edit in the Material Inspector. In the **VertexLit** example the properties look like this:

```
Shader " VertexLit" {
    Properties {
        _Color ("Main Color", Colo
        _SpecColor ("Spec Color"
        _Emission ("Emmisive C
        _Shininess ("Shininess",
        _MainTex ("Base (RGB)",
    }
```



The properties are listed on separate lines within the Properties block. Each property starts with the internal name (**Color**, **MainTex**). After this in parentheses comes the name shown in the inspector and the type of the property. After that, the default value for this property is listed:

The list of possible types are in the Properties Reference. The default value depends on the property type. In the example of a color, the default value should be a four component vector.

We now have our properties defined, and are ready to start writing the actual shader.

The shader body

Before we move on, let's define the basic structure of a shader file.

Different graphic hardware has different capabilities. For example, some graphics cards support fragment programs and others don't; some can lay down four textures per pass while the others can do only two or one; **etc**. To allow you to make full use of whatever hardware your user has, a shader can contain multiple **SubShaders**. When Unity renders a shader, it will go over all subshaders and use the first one that the hardware supports.

```
Shader "Structure Example" {
    Properties { /* ...shader properties... }
    SubShader {
        // ...subshader that requires DX11 / GLES3.1 hardware...
    }
    SubShader {
        // ...subshader that might look worse but runs on anything :)
    }
}
```

This system allows Unity to support all existing hardware and maximize the quality on each one. It does, however, result in some long shaders.

Inside each SubShader block you set the rendering state shared by all passes; and define rendering passes themselves. A complete list of available commands can be found in the SubShader Reference.

Passes

Each subshader is a collection of passes. For each pass, the object geometry is rendered, so there must be at least one pass. Our VertexLit shader has just one pass:

```
// ...snip...
Pass {
    Material {
        Diffuse [_Color]
        Ambient [_Color]
        Shininess [_Shininess]
        Specular [_SpecColor]
        Emission [_Emission]
    }
    Lighting On
    SeparateSpecular On
    SetTexture [_MainTex] {
        constantColor [_Color]
        Combine texture * primary DOUBLE, texture * constant
    }
}
// ...snip...
```

Any commands defined in a pass configures the graphics hardware to render the geometry in a specific way.

In the example above we have a **Material** block that binds our property values to the fixed function lighting material settings. The command **Lighting On** turns on the standard vertex lighting, and **SeparateSpecular On** enables the use of a separate color for the specular highlight.

All of these command so far map very directly to the fixed function OpenGL/Direct3D hardware model. Consult OpenGL red book for more information on this.

The next command, **SetTexture** , is very important. These commands define the textures we want to use and how to mix, combine and apply them in our rendering. **SetTexture** command is followed by the property name of the texture we would like to use (**_MainTex** here) This is followed by a **combiner block** that defines how the texture is applied. The commands in the combiner block are executed for each **pixel** that is rendered on screen.

Within this block we set a constant color value, namely the Color of the Material, **_Color**. We'll use this constant color below.

In the next command we specify how to mix the texture with the color values. We do this with the **Combine** command that specifies how to blend the texture with another one or with a color. Generally it looks like this: **Combine ColorPart, AlphaPart**

Here **ColorPart** and **AlphaPart** define blending of color (RGB) and alpha (A) components respectively. If **AlphaPart** is omitted, then it uses the same blending as **ColorPart**.

In our VertexLit example: **Combine texture * primary DOUBLE, texture * constant**

Here **texture** is the color coming from the current texture (here **_MainTex**). It is multiplied (*) with the **primary** vertex color. Primary color is the vertex lighting color, calculated from the Material values above. Finally, the result is multiplied by two to increase lighting intensity (**DOUBLE**). The alpha value (after the comma) is **texture** multiplied by **constant** value (set with **constantColor** above). Another often used combiner mode is called **previous** (not used in this shader). This is the result of any previous **SetTexture** step, and can be used to combine several textures and/or colors with each other.

Summary

Our VertexLit shader configures standard vertex lighting and sets up the texture combiners so that the rendered lighting intensity is doubled.

We could put more passes into the shader, they would get rendered one after the other. For now, though, that is not nessesary as we have the desired effect. We only need one SubShader as we make no use of any advanced features - this particular shader will work on any graphics card that Unity supports.

The VertexLit shader is one of the most basic shaders that we can think of. We did not use any hardware specific operations, nor did we utilize any of the more special and cool commands that ShaderLab and Cg/HLSL has to offer.

In the next chapter we'll proceed by explaining how to write custom vertex & fragment programs using Cg/HLSL language.

Shaders: vertex and fragment programs

Leave feedback

This tutorial will teach you the basics of how to write vertex and fragment programs in Unity **shaders**. For a basic introduction to **ShaderLab** see the Getting Started tutorial. If you want to write shaders that interact with lighting, read about Surface Shaders instead.

Lets start with a small recap of the general structure of a shader:

```
Shader "MyShaderName"
{
    Properties
    {
        // material properties here
    }
    SubShader // subshader for graphics hardware A
    {
        Pass
        {
            // pass commands ...
        }
        // more passes if needed
    }
    // more subshaders if needed
    FallBack "VertexLit" // optional fallback
}
```

Here at the end we introduce a new command: **FallBack "VertexLit"**. The Fallback command can be used at the end of the shader; it tells which shader should be used if no **SubShaders** from the current shader can run on user's graphics hardware. The effect is the same as including all SubShaders from the fallback shader at the end. For example, if you were to write a fancy normal-mapped shader, then instead of writing a very basic non-normal-mapped subshader for old graphics cards you can just fallback to built-in **VertexLit** shader.

The basic building blocks of the shader are introduced in the first shader tutorial while the full documentation of Properties, SubShaders and Passes are also available.

A quick way of building SubShaders is to use passes defined in other shaders. The command UsePass does just that, so you can reuse shader code in a neat fashion. As an example the following command uses the pass with the name "FORWARD" from the built-in **Specular** shader: **UsePass "Specular/FORWARD"**.

In order for **UsePass** to work, a name must be given to the pass one wishes to use. The Name command inside the pass gives it a name: **Name "MyPassName"**.

Vertex and fragment programs

We described a pass that used just a single texture combine instruction in the first tutorial. Now it is time to demonstrate how we can use vertex and fragment programs in our pass.

When you use vertex and fragment programs (the so called "programmable pipeline"), most of the hardcoded functionality ("fixed function pipeline") in the graphics hardware is switched off. For example, using a vertex program turns off standard 3D

transformations, lighting and texture coordinate generation completely. Similarly, using a fragment program replaces any texture combine modes that would be defined in SetTexture commands; thus SetTexture commands are not needed.

Writing vertex/fragment programs requires a thorough knowledge of 3D transformations, lighting and coordinate spaces - because you have to rewrite the fixed functionality that is built into APIs like OpenGL yourself. On the other hand, you can do much more than what's built in!

Using Cg/HLSL in ShaderLab

Shaders in ShaderLab are usually written in Cg/HLSL programming language. Cg and DX9-style HLSL are for all practical purposes one and the same language, so we'll be using Cg and HLSL interchangeably (see this page for details).

Shader code is written by embedding "Cg/HLSL snippets" in the shader text. Snippets are compiled into low-level shader assembly by the Unity editor, and the final shader that is included in your game's data files only contains this low-level assembly or bytecode, that is platform specific. When you select a shader in the **Project View**, the **Inspector** has a button to show compiled shader code, which might help as a debugging aid. Unity automatically compiles Cg snippets for all relevant platforms (Direct3D 9, OpenGL, Direct3D 11, OpenGL ES and so on). Note that because Cg/HLSL code is compiled by the editor, you can't create shaders from **scripts** at runtime.

In general, snippets are placed inside Pass blocks. They look like this:

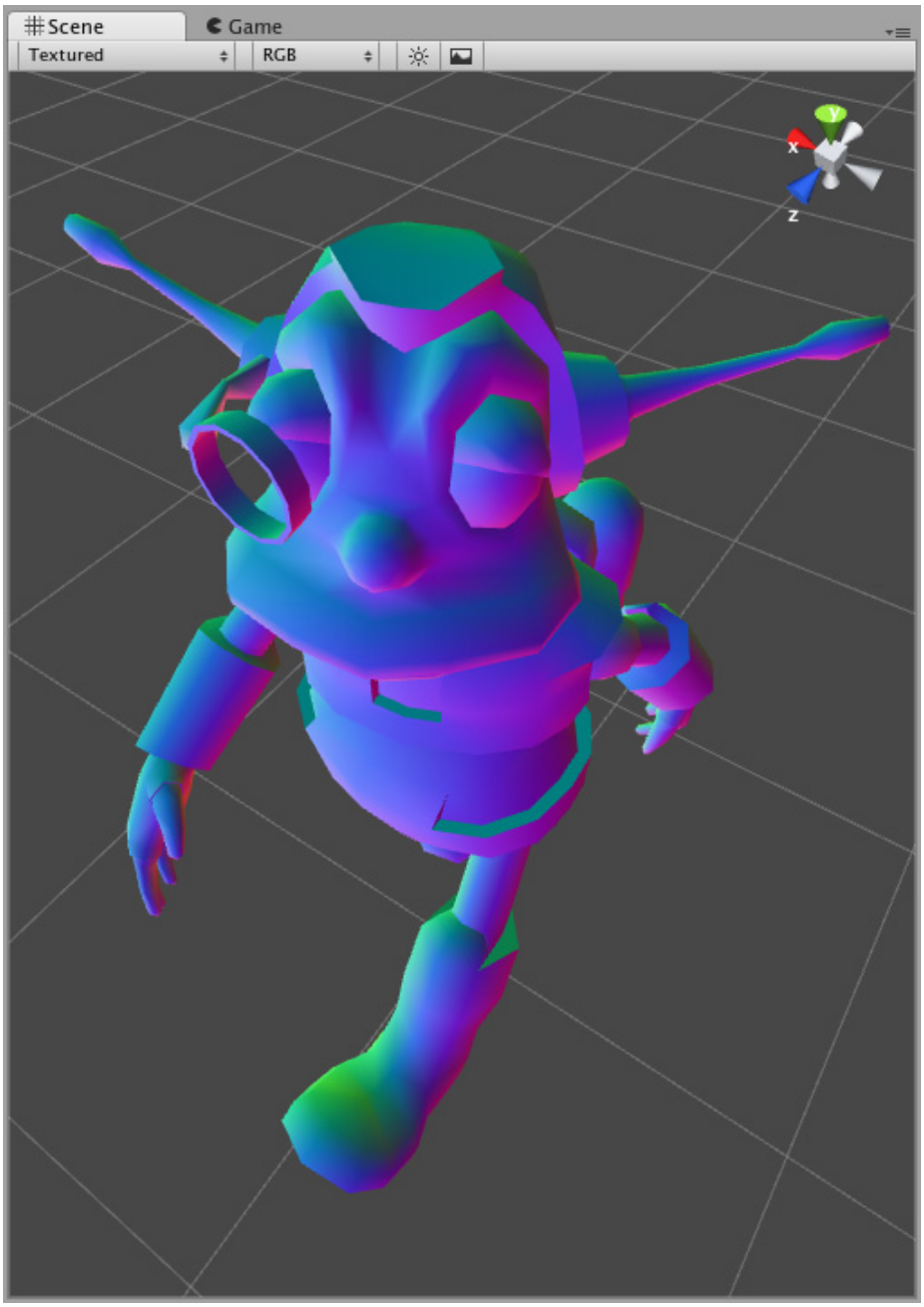
```
Pass {
    // ... the usual pass state setup ...

    CGPROGRAM
    // compilation directives for this snippet, e.g.:
    #pragma vertex vert
    #pragma fragment frag

    // the Cg/HLSL code itself

    ENDCG
    // ... the rest of pass setup ...
}
```

The following example demonstrates a complete shader that renders object normals as colors:

```
Shader "Tutorial/Display Normals" {
    SubShader {
        Pass {

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct v2f {
                float4 pos : SV_POSITION;
```

```
                fixed3 color : COLOR0;
            };

            v2f vert (appdata_base v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.color = v.normal * 0.5 + 0.5;
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                return fixed4 (i.color, 1);
            }
            ENDCG

        }
    }
}
```

When applied on an object it will result in an image like this:



Our “Display Normals” shader does not have any properties, contains a single SubShader with a single Pass that is empty except for the Cg/HLSL code. Let’s dissect the code part by part:

```
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
// ...
ENDCG
```

The whole snippet is written between **CGPROGRAM** and **ENDCG** keywords. At the start compilation directives are given as **#pragma** statements:

#pragma vertex name tells that the code contains a vertex program in the given function (**vert** here).
#pragma fragment name tells that the code contains a fragment program in the given function (**frag** here).

Following the compilation directives is just plain Cg/HLSL code. We start by including a built-in include file:

```
#include "UnityCG.cginc"
```

The **UnityCG.cginc** file contains commonly used declarations and functions so that the shaders can be kept smaller (see shader include files page for details). Here we'll use **appdata_base** structure from that file. We could just define them directly in the shader and not include the file of course.

Next we define a "vertex to fragment" structure (here named **v2f**) - what information is passed from the vertex to the fragment program. We pass the position and color parameters. The color will be computed in the vertex program and just output in the fragment program.

We proceed by defining the vertex program - **vert** function. Here we compute the position and output input normal as a color: o.color = v.normal * 0.5 + 0.5;

Normal components are in –1..1 range, while colors are in 0..1 range, so we scale and bias the normal in the code above. Next we define a fragment program - **frag** function that just outputs the calculated color and 1 as the alpha **component**:

```
fixed4 frag (v2f i) : SV_Target
{
    return fixed4 (i.color, 1);
}
```

That's it, our shader is finished! Even this simple shader is very useful to visualize **mesh** normals.

Of course, this shader does not respond to lights at all, and that's where things get a bit more interesting; read about Surface Shaders for details.

Using shader properties in Cg/HLSL code

When you define properties in the shader, you give them a name like **_Color** or **_MainTex**. To use them in Cg/HLSL you just have to define a variable of a matching name and type. See properties in shader programs page for details.

Here is a complete shader that displays a texture modulated by a color. Of course, you could easily do the same in a texture combiner call, but the point here is just to show how to use properties in Cg:

```
Shader "Tutorial/Textured Colored" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0.5)
        _MainTex ("Texture", 2D) = "white" { }
    }
    SubShader {
        Pass {

        CGPROGRAM
        #pragma vertex vert
```

```
        #pragma fragment frag

        #include "UnityCG.cginc"

        fixed4 _Color;
        sampler2D _MainTex;

        struct v2f {
            float4 pos : SV_POSITION;
            float2 uv : TEXCOORD0;
        };

        float4 _MainTex_ST;

        v2f vert (appdata_base v)
        {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex);
            o.uv = TRANSFORM_TEX (v.texcoord, _MainTex);
            return o;
        }

        fixed4 frag (v2f i) : SV_Target
        {
            fixed4 texcol = tex2D (_MainTex, i.uv);
            return texcol * _Color;
        }
        ENDCG

        }
    }
}
```

The structure of this shader is the same as in the previous example. Here we define two properties, namely **_Color** and **_MainTex**. Inside Cg/HLSL code we define corresponding variables:

```
fixed4 _Color;
sampler2D _MainTex;
```

See Accessing Shader Properties in Cg/HLSL for more information.

The vertex and fragment programs here don't do anything fancy; vertex program uses the **TRANSFORM_TEX** macro from UnityCG.cginc to make sure texture scale and offset is applied correctly, and fragment program just samples the texture and multiplies by the color property.

Summary

We have shown how custom shader programs can be written in a few easy steps. While the examples shown here are very simple, there's nothing preventing you to write arbitrarily complex shader programs! This can help you to take the full advantage of Unity and achieve optimal **rendering** results.

The complete ShaderLab reference manual is here, and more examples in vertex and fragment shader examples page. We also have a forum for shaders at forum.unity3d.com so go there to get help with your shaders! Happy programming, and enjoy the power of Unity and ShaderLab.

Scriptable Render Pipeline

Leave feedback

The Scriptable Render Pipeline (SRP) is an alternative to the Unity built-in pipeline. With the SRP, you can control and tailor **rendering** via C# **scripts**. This way, you can either slightly modify or completely build and customize the render pipeline to your needs.

The SRP gives you more granularity and customization options than the built-in Unity render pipeline. And you can use one of the pre-built SRPs to target your specific needs.

Using an SRP is different from using the built-in Unity render pipeline. For example, the pre-built SRPs come with a new **Shader** library, because the built-in Unity Shaders do not work with SRPs.

Note: This is a preview feature and is subject to change. Any scripts that use this feature may need updating in a future release. Do not rely on this feature for full-scale production until it is no longer in preview.

Pre-built SRPs

Unity has built two Scriptable Render Pipelines that use the SRP framework: the High Definition Render Pipeline (HDRP) and the Lightweight Render Pipeline (LWRP). Each render pipeline targets a specific set of use-case scenarios and hardware needs. The pre-built render pipelines are available as templates for new Projects.

Both of these render pipelines are delivered as packages, and include the Shader Graph and Post Processing packages.

Note: Projects made using HDRP are not compatible with the Lightweight Render Pipeline or the built-in Unity rendering pipeline. Before you start development, you must decide which render pipeline to use in your Project.

High Definition Render Pipeline

The HDRP targets high-end hardware like consoles and PCs. With the HDRP, you're able to achieve realistic graphics in demanding scenarios. The HDRP uses Compute Shader technology and therefore requires compatible GPU hardware.

Use HDRP for AAA quality games, automotive demos, architectural applications and anything that favors high-fidelity graphics over performance. HDRP uses physically-based Lighting and Materials, and supports both Forward and Deferred rendering.

The High Definition Render Pipeline is provided in a single template.

For more information on HDRP, see the HDRP overview on the SRP GitHub Wiki.

Lightweight Render Pipeline

You can use the LWRP across a wide range of hardware. The technology is scalable to mobile platforms, and you can also use it for higher-end consoles and PCs. You're able to achieve quick rendering at a high quality. LWRP uses simplified, physically based Lighting and Materials.

The LWRP uses single-pass **forward rendering**. Use this pipeline to get optimized real-time performance on several platforms.

The Lightweight Render Pipeline is available via two templates: LWRP and LWRP-VR. The LWRP-VR comes with pre-enabled settings specifically for VR.

Note: Built-in and custom Lit Shaders do not work with the Lightweight Render Pipeline. Instead, LWRP has a new set of **standard shaders**. If you upgrade a current Project to LWRP, you can upgrade built-in shaders to the new ones.

Fore more information on LWRP, see the LWRP overview on the SRP GitHub Wiki.

Setting up a Scriptable Render Pipeline

There are several different ways to install SRP, HDRP and LWRP: If you want to use one of the pre-built SRPs in a new or current Project, continue reading this section. If you are an advanced user, and you want to modify the SRP scripts to directly, see Creating a custom SRP.

Using a pre-built SRP with a new Project

If you want to use the HDRP or the LWRP in a new Project, and you don't need to customise the render pipeline, you can create a new Project using Templates.

To create a Project using Templates:

Open Unity. On the Home page, click **New** to start a new Project.
Under **Template**, select one of the render pipeline templates.
Click **Create Project**. Unity automatically creates a new Project for you, complete with all the functionalities of the included built-in pipeline.
For more information on using Templates, see Project Templates.

Installing the latest SRP into an existing Project

You can download and install the latest version of HDRP, LWRP or SRP to your existing Project via the Package Manager system.

Switching to an SRP from an existing Project consumes a lot of time and resources. HDRP, LWRP and custom SRPs all use custom shaders. They are not compatible with the built-in Unity shaders. You will have to manually change or convert many elements. Instead, consider starting a new Project with your SRP of choice.

To install an SRP into an existing Project:

In Unity, open your Project.
In the Unity menu bar, go to **Window** > **Package Manager** to open the **Package Manager** window.
Select the **All** tab. This tab displays the list of available packages for the version of Unity that you are currently running.
Select the render pipeline that you want to use from the list of packages. In the top right corner of the window, click **Install**. This installs the render pipeline directly in to your Project.

Configuring and using a pre-built render pipeline

Before you can use either of the pre-built render pipelines, you must create a Render Pipeline **Asset** and add it to your **Project settings**. The following sections explain how to create the SRP **Asset** and how to add it to a HDRP or LWRP.

Creating Scriptable Render Pipeline Assets

To properly use your chosen SRP, you must create a Scriptable Render Pipeline Asset.

The Scriptable Render Pipeline Asset controls the global rendering quality settings of your Project and creates the rendering pipeline instance. The rendering pipeline instance contains intermediate resources and the render pipeline implementation.

You can create multiple Pipeline Assets to store settings for different platforms or for different testing environments.

To create a Render Pipeline Asset:

In the Editor, go to the Project window, and navigate to a directory outside of the Scriptable Render Pipeline Folder.

Right-click in the Project window, and select **Create > Render Pipeline**.
Select either **High Definition** or **Lightweight**, depending on your needs. Click **Render Pipeline/Pipeline Asset**.

Configuring and using HDRP

To use HDRP, you must edit your Project's **Player** and **Graphics** settings as follows:

In the Editor, go to **Edit** > **Project Settings > Player**. In the **Color Space** dropdown, select **Linear**. HDRP does not support Gamma lighting.
In the Project window, navigate to a directory outside of the *Scriptable Render Pipeline* folder.
Right-click the Project window, and select **Create > Render Pipeline > High Definition > Render Pipeline**.
Navigate to **Edit > Project Settings > Graphics**.
In the **Render Pipeline Settings** field, add the High Definition Render Pipeline Asset you created earlier.
Tip: Always store your High Definition Render Pipeline Asset outside of the Scriptable Render Pipeline folder. This ensures that your HDRP settings are not lost when merging new changes from the SRP GitHub repository.

Configuring and using LWRP

To use the LWRP, you must edit your Project's **Graphics** settings as follows:

In the Project window, navigate to a directory outside of the *Scriptable Render Pipeline* folder.
Right-click in the Project window, and select **Create > Render Pipeline > Lightweight > Pipeline Asset**.
Navigate to **Edit > Project Settings > Graphics**.
In the *_Render Pipeline Settings* field, add the Lightweight Render Pipeline Asset you created earlier.
Tip: Always store your new Render Pipeline Asset outside of the *Scriptable Render Pipeline* folder. This ensures that your Lightweight settings are not lost when merging new changes from the SRP GitHub repository.

Creating a custom SRP

The Scriptable Render Pipelines are available in an open Project on GitHub. You can clone an SRP and make modifications in your local version.

To configure local script files for a new or existing Unity Project:

Create a clone of the SRP repository. Place the clone in the root of the Project directory, next to the *Assets* folder. For information on cloning repositories, see the GitHub help on Cloning a repository.

Checkout a branch that is compatible with your version of Unity. Use the command `git checkout`, and type the branch name.

Use the `git submodule update --init` command to find and initialize all submodules related to the SRP.

In your Project manifest, update `dependencies` to so that they point to the SRP packages. To read more about Project manifests, see the Package Manager documentation. Here is an example of what your script should look like:

```
{

   "dependencies": {
       "com.unity.postprocessing": "file:../ScriptableRenderPipeline/com.unity.postproc

       "com.unity.render-pipelines.core": "file:../ScriptableRenderPipeline/com.unity.r
```

```
        "com.unity.shadergraph": "file:../ScriptableRenderPipeline/com.unity.shadergraph

        "com.unity.render-pipelines.lightweight": "file:../ScriptableRenderPipeline/com.

    }
}
```

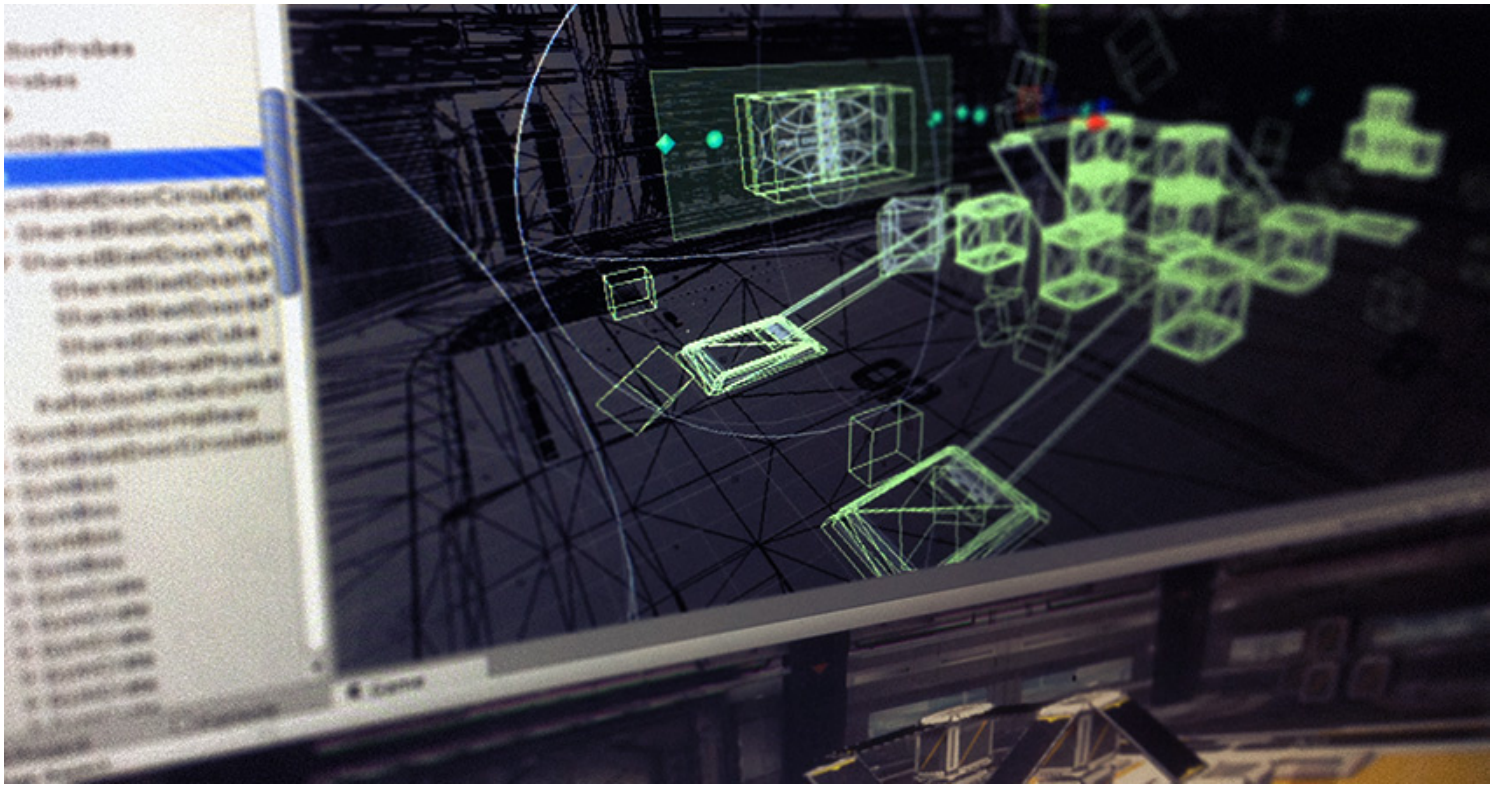
Open your Project in Unity. Your packages are now installed locally. When you open the Project solution in an integrated development environment, you can debug and modify the script directly.

2018–07–03 Page published with editorial review

Scriptable Render Pipline added in 2018.1

Physics

Leave feedback



To have convincing physical behaviour, an object in a game must accelerate correctly and be affected by **collisions**, gravity and other forces. Unity's built-in **physics engines** provide **components** that handle the physical simulation for you. With just a few parameter settings, you can create objects that behave passively in a realistic way (ie, they will be moved by collisions and falls but will not start moving by themselves). By controlling the physics from **scripts**, you can give an object the dynamics of a vehicle, a machine, or even a piece of fabric. This section gives an overview of the main physics components in Unity, with links for further reading.

Note: there are actually two separate physics engines in Unity: one for 3D physics, and one for 2D physics. The main concepts are identical between the two engines (except for the extra dimension in 3D), but they are implemented using different components. For example, there is **Rigidbody** component for 3D physics and an analogous **Rigidbody 2D** for 2D physics.

Related tutorials: Physics; Physics Best Practices

See the Knowledge Base Physics section for troubleshooting, tips and tricks.

Physics Overview

Leave feedback

These pages briefly describe the main physics **components** available in Unity, and give details of their usage and links for further reading.

Rigidbody overview

Leave feedback

A **Rigidbody** is the main **component** that enables physical behaviour for a **GameObject**. With a Rigidbody attached, the object will immediately respond to gravity. If one or more **Collider** components are also added, the **GameObject** is moved by incoming **collisions**.

Since a Rigidbody component takes over the movement of the GameObject it is attached to, you shouldn't try to move it from a script by changing the Transform properties such as position and rotation. Instead, you should apply **forces** to push the GameObject and let the **physics engine** calculate the results.

There are some cases where you might want a GameObject to have a Rigidbody without having its motion controlled by the physics engine. For example, you may want to control your character directly from script code but still allow it to be detected by triggers (see *Triggers* under the Colliders topic). This kind of non-physical motion produced from a script is known as *kinematic* motion. The Rigidbody component has a property called **Is Kinematic** which removes it from the control of the physics engine and allow it to be moved kinematically from a script. It is possible to change the value of **Is Kinematic** from a script to allow physics to be switched on and off for an object, but this comes with a performance overhead and should be used sparingly.

See the Rigidbody and Rigidbody 2D reference pages for further details about the settings and scripting options for these components.

Sleeping

When a Rigidbody is moving slower than a defined minimum linear or rotational speed, the physics engine assumes it has come to a halt. When this happens, the GameObject does not move again until it receives a collision or force, and so it is set to "sleeping" mode. This optimisation means that no processor time is spent updating the Rigidbody until the next time it is "awoken" (that is, set in motion again).

For most purposes, the sleeping and waking of a Rigidbody component happens transparently. However, a GameObject might fail to wake up if a Static Collider (that is, one without a Rigidbody) is moved into it or away from it by modifying the **Transform** position. This might result, say, in the Rigidbody GameObject hanging in the air when the floor has been moved out from beneath it. In cases like this, the GameObject can be woken explicitly using the `WakeUp` function. See the Rigidbody and Rigidbody 2D component pages for more information about sleeping.

Colliders

Leave feedback

Collider components define the shape of an object for the purposes of physical **collisions**. A collider, which is invisible, need not be the exact same shape as the object's **mesh** and in fact, a rough approximation is often more efficient and indistinguishable in gameplay.

The simplest (and least processor-intensive) colliders are the so-called *primitive* collider types. In 3D, these are the Box Collider, Sphere Collider and Capsule Collider. In 2D, you can use the Box Collider 2D and Circle Collider 2D. Any number of these can be added to a single object to create *compound colliders*.

With careful positioning and sizing, compound colliders can often approximate the shape of an object quite well while keeping a low processor overhead. Further flexibility can be gained by having additional colliders on child objects (eg, boxes can be rotated relative to the local axes of the parent object). When creating a compound collider like this, there should only be one **Rigidbody** component, placed on the root object in the hierarchy.

Note, that primitive colliders will not work correctly with shear transforms - that means that if you use a combination of rotations and non-uniform scales in the Transform hierarchy so that the resulting shape would no longer match a primitive shape, the primitive collider will not be able to represent it correctly.

There are some cases, however, where even compound colliders are not accurate enough. In 3D, you can use Mesh Colliders to match the shape of the object's mesh exactly. In 2D, the Polygon Collider 2D will generally not match the shape of the **sprite** graphic perfectly but you can refine the shape to any level of detail you like. These colliders are much more processor-intensive than primitive types, however, so use them sparingly to maintain good performance. Also, a mesh collider will normally be unable to collide with another mesh collider (ie, nothing will happen when they make contact). You can get around this in some cases by marking the mesh collider as **Convex** in the **inspector**. This will generate the collider shape as a "convex hull" which is like the original mesh but with any undercuts filled in. The benefit of this is that a convex mesh collider *can* collide with other mesh colliders so you may be able to use this feature when you have a moving character with a suitable shape. However, a good general rule is to use mesh colliders for **scene** geometry and approximate the shape of moving objects using compound primitive colliders.

Colliders can be added to an object without a Rigidbody component to create floors, walls and other motionless elements of a scene. These are referred to as **static** colliders. In general, you should not reposition static colliders by changing the Transform position since this will impact heavily on the performance of the **physics engine**. Colliders on an object that *does* have a Rigidbody are known as *dynamic* colliders. Static colliders can interact with dynamic colliders but since they don't have a Rigidbody, they will not move in response to collisions.

The reference pages for the various collider types linked above have further information about their properties and uses.

Physics materials

When colliders interact, their surfaces need to simulate the properties of the material they are supposed to represent. For example, a sheet of ice will be slippery while a rubber ball will offer a lot of friction and be very bouncy. Although the shape of colliders is not deformed during collisions, their friction and bounce can be configured using **Physics Materials**. Getting the parameters just right can involve a bit of trial and error but an ice material, for example will have zero (or very low) friction and a rubber material with have high friction and near-perfect bounciness. See the reference pages for Physic Material and Physics Material 2D for further details on the available parameters. Note that for historical reasons, the 3D **asset** is actually called **Physic Material** (*without* the S) but the 2D equivalent is called **Physics Material 2D** (*with* the S).

Triggers

The scripting system can detect when collisions occur and initiate actions using the `OnCollisionEnter` function. However, you can also use the physics engine simply to detect when one collider enters the space of another without creating a collision. A collider configured as a **Trigger** (using the **Is Trigger** property) does not behave as a solid object and will simply

allow other colliders to pass through. When a collider enters its space, a trigger will call the `OnTriggerEnter` function on the trigger object's **scripts**.

Script actions taken on collision

When collisions occur, the physics engine calls functions with specific names on any scripts attached to the objects involved. You can place any code you like in these functions to respond to the collision event. For example, you might play a crash sound effect when a car bumps into an obstacle.

On the first physics update where the collision is detected, the `OnCollisionEnter` function is called. During updates where contact is maintained, `OnCollisionStay` is called and finally, `OnCollisionExit` indicates that contact has been broken. Trigger colliders call the analogous `OnTriggerEnter`, `OnTriggerStay` and `OnTriggerExit` functions. Note that for 2D physics, there are equivalent functions with **2D** appended to the name, eg, `OnCollisionEnter2D`. Full details of these functions and code samples can be found on the Script Reference page for the MonoBehaviour class.

With normal, non-trigger collisions, there is an additional detail that at least one of the objects involved must have a non-kinematic Rigidbody (ie, *Is Kinematic* must be switched off). If both objects are kinematic Rigidbodies then `OnCollisionEnter`, **etc**, will not be called. With trigger collisions, this restriction doesn't apply and so both kinematic and non-kinematic Rigidbodies will prompt a call to `OnTriggerEnter` when they enter a trigger collider.

Collider interactions

Colliders interact with each other differently depending on how their Rigidbody components are configured. The three important configurations are the *Static Collider* (ie, no Rigidbody is attached at all), the *Rigidbody Collider* and the *Kinematic Rigidbody Collider*.

Static Collider

This is a **GameObject** that has a Collider but no Rigidbody. Static colliders are used for level geometry which always stays at the same place and never moves around. Incoming rigidbody objects will collide with the static collider but will not move it.

The physics engine assumes that static colliders never move or change and can make useful optimizations based on this assumption. Consequently, static colliders should not be disabled/enabled, moved or scaled during gameplay. If you do change a static collider then this will result in extra internal recomputation by the physics engine which causes a major drop in performance. Worse still, the changes can sometimes leave the collider in an undefined state that produces erroneous physics calculations. For example a raycast against an altered Static Collider could fail to detect it, or detect it at a random position in space. Furthermore, Rigidbodies that are hit by a moving static collider will not necessarily be "awoken" and the static collider will not apply any friction. For these reasons, only colliders that are Rigidbodies should be altered. If you want a collider object that is not affected by incoming rigidbodies but can still be moved from a script then you should attach a *Kinematic* Rigidbody component to it rather than no Rigidbody at all.

Rigidbody Collider

This is a GameObject with a Collider and a normal, non-kinematic Rigidbody attached. Rigidbody colliders are fully simulated by the physics engine and can react to collisions and forces applied from a script. They can collide with other objects (including static colliders) and are the most commonly used Collider configuration in games that use physics.

Kinematic Rigidbody Collider

This is a GameObject with a Collider and a *kinematic* Rigidbody attached (ie, the *IsKinematic* property of the Rigidbody is enabled). You can move a kinematic rigidbody object from a script by modifying its **Transform Component** but it will not respond to collisions and forces like a non-kinematic rigidbody. Kinematic rigidbodies should be used for colliders that can be moved or disabled/enabled occasionally but that should otherwise behave like static colliders. An example of this is a sliding

door that should normally act as an immovable physical obstacle but can be opened when necessary. Unlike a static collider, a moving kinematic rigidbody will apply friction to other objects and will “wake up” other rigidbodies when they make contact.

Even when immobile, kinematic rigidbody colliders have different behavior to static colliders. For example, if the collider is set to as a trigger then you also need to add a rigidbody to it in order to receive trigger events in your script. If you don’t want the trigger to fall under gravity or otherwise be affected by physics then you can set the *IsKinematic* property on its rigidbody.

A Rigidbody component can be switched between normal and kinematic behavior at any time using the *IsKinematic* property.

A common example of this is the “ragdoll” effect where a character normally moves under animation but is thrown physically by an explosion or a heavy collision. The character’s limbs can each be given their own Rigidbody component with *IsKinematic* enabled by default. The limbs will move normallly by animation until *IsKinematic* is switched off for all of them and they immediately behave as physics objects. At this point, a collision or explosion force will send the character flying with its limbs thrown in a convincing way.

Collision action matrix

When two objects collide, a number of different script events can occur depending on the configurations of the colliding objects’ rigidbodies. The charts below give details of which event functions are called based on the components that are attached to the objects. Some of the combinations only cause one of the two objects to be affected by the collision, but the general rule is that physics will not be applied to an object that doesn’t have a Rigidbody component attached.

Collision detection occurs and messages are sent upon collision

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		Y				
Rigidbody Collider	Y	Y	Y			
Kinematic Rigidbody Collider		Y				
Static Trigger Collider						
Rigidbody Trigger Collider						
Kinematic Rigidbody Trigger Collider						

Trigger messages are sent upon collision

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider					Y	Y
Rigidbody Collider				Y	Y	Y
Kinematic Rigidbody Collider				Y	Y	Y
Static Trigger Collider		Y	Y		Y	Y
Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y
Kinematic Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y

Joints

Leave feedback

You can attach one **rigidbody** object to another or to a fixed point in space using a **Joint component**. Generally, you want a joint to allow at least *some* freedom of motion and so Unity provides different Joint components that enforce different restrictions.

For example, a Hinge Joint allows rotation around a specific point and axis while a Spring Joint keeps the objects apart but lets the distance between them stretch slightly.

2D joint components have **2D** at the end of the name, eg, Hinge Joint 2D. See Joints 2D for a summary of the 2D joints and useful background information.

Joints also have other options that can enabled for specific effects. For example, you can set a joint to break when the force applied to it exceeds a certain threshold. Some joints also allow a **drive force** to occur between the connected objects to set them in motion automatically.

See each joint reference page for the Joint classes and for further information about their properties.

Character Controllers

Leave feedback

The character in a first- or third-person game will often need some collision-based physics so that it doesn't fall through the floor or walk through walls. Usually, though, the character's acceleration and movement will not be physically realistic, so it may be able to accelerate, brake and change direction almost instantly without being affected by momentum.

In 3D physics, this type of behaviour can be created using a **Character Controller**. This **component** gives the character a simple, capsule-shaped **collider** that is always upright. The controller has its own special functions to set the object's speed and direction but unlike true colliders, a **rigidbody** is not needed and the momentum effects are not realistic.

A character controller cannot walk through static colliders in a **scene**, and so will follow floors and be obstructed by walls. It can push rigidbody objects aside while moving but will not be accelerated by incoming **collisions**. This means that you can use the standard 3D colliders to create a **scene** around which the controller will walk but you are not limited by realistic physical behaviour on the character itself.

You can find out more about character controllers on the reference page.

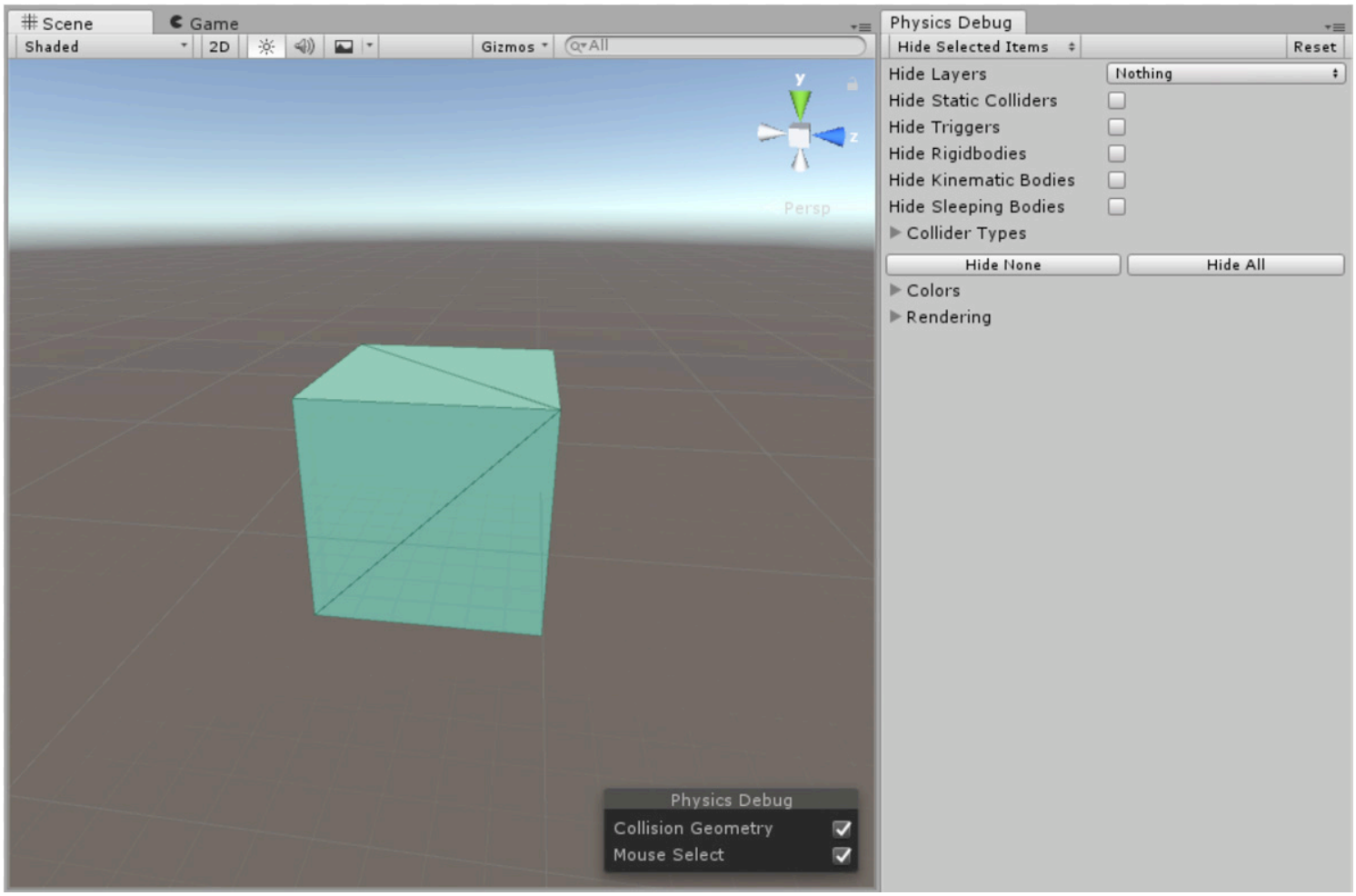
Physics Debug Visualization

Leave feedback

Physics Debug Visualiser allows you to quickly inspect the Collider geometry in your Scene, and profile common physics-based scenarios. It provides a visualisation of which **GameObjects** should and should not collide with each other. This is particularly useful when there are many Colliders in your Scene, or if the Render and Collision Meshes are out of sync.

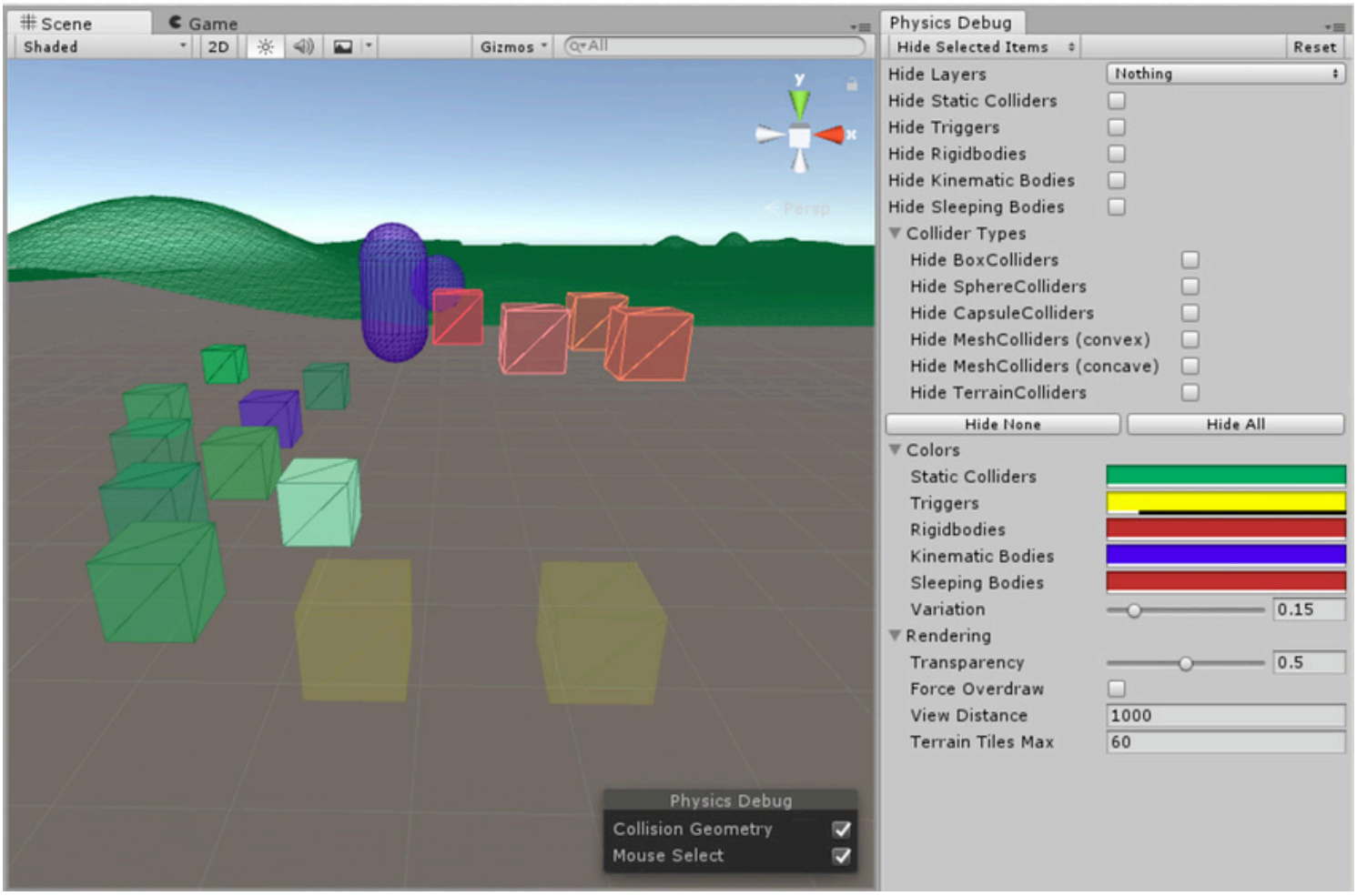
For further guidance on improving physics performance in your project, see documentation on the Physics Profiler.

To open the Physics Debug window in the Unity Editor, go to **Window** > **Analysis** > **Physics Debugger**.

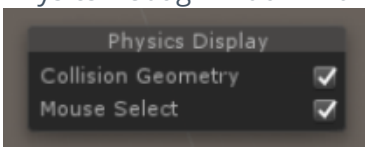


Default Physics Debug Visualization settings and a Cube primitive

From this window, you can customize visual settings and specify the types of GameObjects you want to see or hide in the visualizer.



Physics Debug window with fold-out options and overlay panel



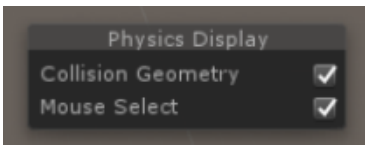
The Physics Debug overlay panel

Hide Selected Items is the default mode. This means that every item appears in the visualizer, and you need to tick the checkbox for each item to hide it. To change this to **Show Selected Items**, use the drop-down at the top of the window. This means that no items appear in the visualizer, and you need to tick the checkbox for each item to display it.

Property	Function
Reset	Click this button to reset the Physics Debug window back to default settings.
Hide Layers	Use the dropdown menu to determine whether or not to display Colliders from the selected Layers.
Hide Static Colliders	Tick this checkbox to remove static Colliders (Colliders with no Rigidbody component) from the visualization.
Hide Triggers	Tick this checkbox to remove Colliders that are also Triggers from the visualization.
Hide Rigidbodies	Tick this checkbox to remove Rigidbody components from the visualization.
Hide Kinematic Bodies	Tick this checkbox to remove Colliders with **Kinematic** Rigidbody **components** (which are not controlled by the physics engine) from the visualization. See documentation on Rigidbody components for more details.
Hide Sleeping Bodies	Tick this checkbox to remove Colliders with **Sleeping Rigidbody** components (which are not currently engaging with the physics engine) from the visualization. See documentation on Rigidbody components: Sleeping for more details.
Collider Types	Use these options to remove specific Collider types from the physics visualization.

Property	Function
Hide BoxColliders	Tick this checkbox to remove Box Colliders from the visualization.
Hide SphereColliders	Tick this checkbox to remove Sphere Colliders from the visualization.
Hide CapsuleColliders	Tick this checkbox to remove Capsule Colliders from the visualization.
Hide MeshColliders (convex)	Tick this checkbox to remove **convex** Mesh Colliders from the visualization.
Hide MeshColliders (concave)	Tick this checkbox to remove **concave** Mesh Colliders from the visualization.
Hide TerrainColliders	Tick this checkbox to remove Terrain Colliders from the visualization.
Hide None	Click **Hide None** to clear all filtering criteria and display all Collider types in the visualization.
Hide All	Click **Hide All** to enable all filters and remove all Collider types from the visualization.
Colors	Use these settings to define how Unity displays physics components in the visualization.
Static Colliders	Use this color selector to define which color indicates static Colliders (Colliders with no Rigidbody component) in the visualization.
Triggers	Use this color selector to define which color indicates Colliders that are also Triggers in the visualization.
Rigidbodies	Use this color selector to define which color indicates Rigidbody components in the visualization.
Kinematic Bodies	Use this color selector to define which color indicates **Kinematic** Rigidbody components (which are not controlled by the physics engine) from the visualization. See documentation on Rigidbody components for more details.
Sleeping Bodies	Use this color selector to define which color indicates **Sleeping** Rigidbody components (which are not currently engaging with the physics engine) from the visualization. See documentation on Rigidbody components: Sleeping for more details.
Variation	Use the slider to set a value between 0 and 1. This defines how much of your chose color blends with a random color. Use this to visually separate Colliders by color, and to see the structure of the GameObjects.
Rendering	Use these settings to define how Unity renders and displays the physics visualization.
Transparency	Use the slider to set the value between 0 and 1. This defines the transparency of drawn **collision** geometry in the visualization.
Force Overdraw	Normal render geometry can sometimes obscure Colliders (for example, a Mesh Collider plane underneath a floor).Tick the **Force Overdraw** checkbox to make the visualization renderer draw Collider geometry on top of render geometry.
View Distance	Use this to set the view distance of the visualization.
Terrain Tiles Max	Use this to set the maximum number of Terrain tiles in the visualization.

The overlay panel has further options:



The Physics Debug overlay panel

Property	Function

Property	Function
Collision Geometry	Tick this checkbox to enable collision geometry visualization.
Mouse Select	Tick this checkbox to enable mouse-over highlighting and mouse selection. This can be useful if you have large GameObjects obstructing each other in the visualizer.

Profiling

You can use Physics Debug to profile and troubleshoot physics activity in your game. You can customize which types of Colliders or Rigidbody components you can see in the visualiser, to help you find the source of activity. The two most helpful are:

To see active Rigidbody components only: To see only the Rigidbody components that are active and therefore using CPU/GPU resources, tick **Hide Static Colliders** and **Hide Sleeping Bodies**.

To see non-convex Mesh Colliders only: Non-convex (triangle-based) Mesh Colliders tend to generate the most contacts when their attached Rigidbody components are very near a collision with another Rigidbody or Collider. To visualize only the non-convex Mesh Colliders, set the window to **Show Selected Items** mode, click the **Select None** button, then tick the **Show MeshColliders (concave)** checkbox.

Scripting API Reference

See Unity Scripting API Reference for:

PhysicsDebugWindow

PhysicsVisualizationSettings

2017–06–01 Page amended with editorial review

Physics Debug Visualization is a new feature in Unity 5.6

3D Physics Reference

Leave feedback

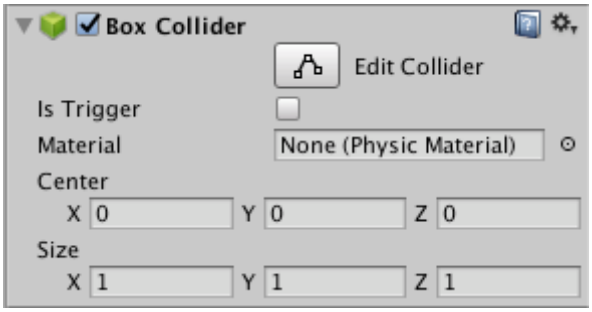
This section gives details of the **components** used with 3D physics. See Physics 2D Reference for the equivalent 2D components.

Box Collider

Leave feedback

SWITCH TO SCRIPTING

The **Box Collider** is a basic cube-shaped **collision** primitive.



Properties

Property:	*Function:*
Is Trigger	If enabled, this **Collider** is used for triggering events, and is ignored by the **physics engine**.
Material	Reference to the Physics Material that determines how this Collider interacts with others.
Center	The position of the Collider in the object's local space.
Size	The size of the Collider in the X, Y, Z directions.

Details

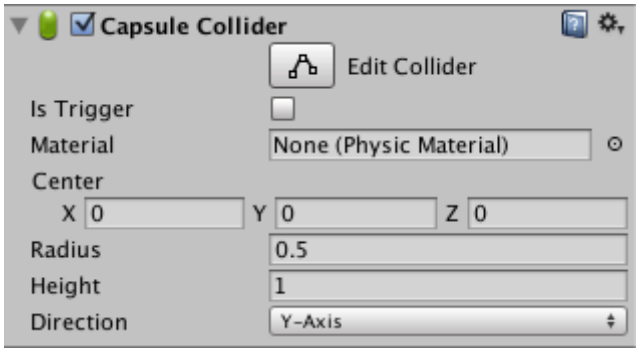
Box colliders are obviously useful for anything roughly box-shaped, such as a crate or a chest. However, you can use a thin box as a floor, wall or ramp. The box shape is also a useful element in a compound collider.

Capsule Collider

Leave feedback

SWITCH TO SCRIPTING

The **Capsule Collider** is made of two half-spheres joined together by a cylinder. It is the same shape as the Capsule primitive.

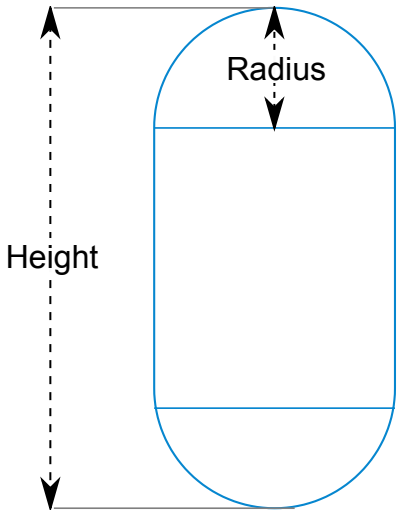


Properties

Property:	*Function:*
Is Trigger	If enabled, this **Collider** is used for triggering events, and is ignored by the **physics engine**.
Material	Reference to the Physics Material that determines how this Collider interacts with others.
Center	The position of the Collider in the object's local space.
Radius	The radius of the Collider's local width.
Height	The total height of the Collider.
Direction	The axis of the capsule's lengthwise orientation in the object's local space.

Details

You can adjust the Capsule Collider's **Radius** and **Height** independently of each other. It is used in the Character Controller and works well for poles, or can be combined with other Colliders for unusual shapes.



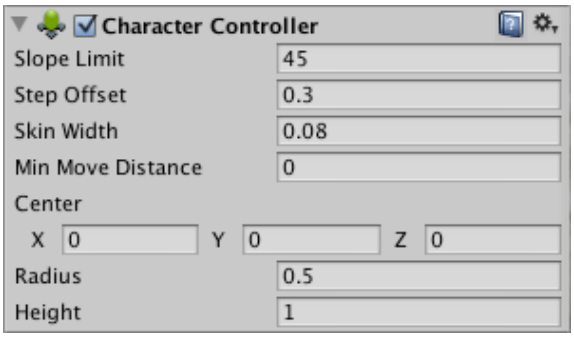
A standard Capsule Collider

Character Controller

Leave feedback

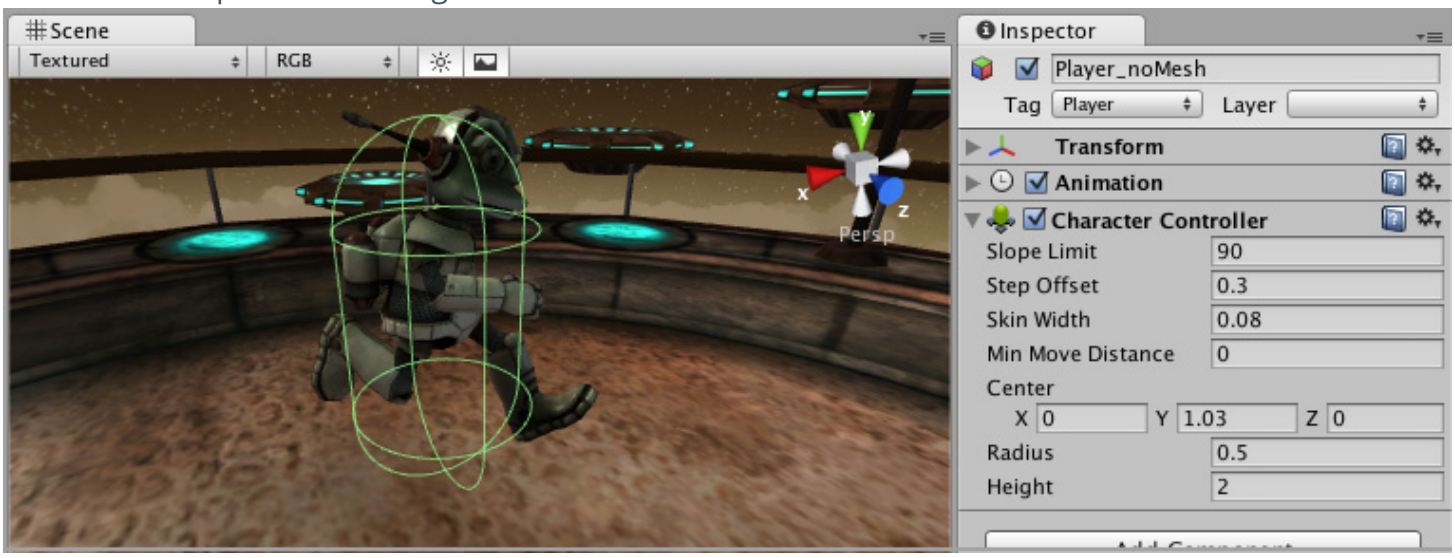
SWITCH TO SCRIPTING

The **Character Controller** is mainly used for third-person or first-person player control that does not make use of **Rigidbody** physics.



Properties

Property:	*Function:*
Slope Limit	Limits the collider to only climb slopes that are less steep (in degrees) than the indicated value.
Step Offset	The character will step up a stair only if it is closer to the ground than the indicated value. This should not be greater than the Character Controller's height or it will generate an error.
Skin width	Two colliders can penetrate each other as deep as their Skin Width. Larger Skin Widths reduce jitter. Low Skin Width can cause the character to get stuck. A good setting is to make this value 10% of the Radius.
Min Move Distance	If the character tries to move below the indicated value, it will not move at all. This can be used to reduce jitter. In most situations this value should be left at 0.
Center	This will offset the Capsule Collider in world space, and won't affect how the Character pivots.
Radius	Length of the Capsule Collider's radius. This is essentially the width of the collider.
Height	The Character's **Capsule Collider** height. Changing this will scale the collider along the Y axis in both positive and negative directions.



The Character Controller

Details

The traditional Doom-style first person controls are not physically realistic. The character runs 90 miles per hour, comes to a halt immediately and turns on a dime. Because it is so unrealistic, use of Rigidbodies and physics to create this behavior is impractical and will feel wrong. The solution is the specialized Character Controller. It is simply a capsule shaped **Collider** which can be told to move in some direction from a script. The Controller will then carry out the movement but be constrained by **collisions**. It will slide along walls, walk up stairs (if they are lower than the **Step Offset**) and walk on slopes within the **Slope Limit**.

The Controller does not react to forces on its own and it does not automatically push Rigidbodies away.

If you want to push Rigidbodies or objects with the Character Controller, you can apply forces to any object that it collides with via the **OnControllerColliderHit()** function through scripting.

On the other hand, if you want your player character to be affected by physics then you might be better off using a Rigidbody instead of the Character Controller.

Fine-tuning your character

You can modify the **Height** and **Radius** to fit your Character's **mesh**. It is recommended to always use around 2 meters for a human-like character. You can also modify the **Center** of the capsule in case your pivot point is not at the exact center of the Character.

Step Offset can affect this too, make sure that this value is between 0.1 and 0.4 for a 2 meter sized human.

Slope Limit should not be too small. Often using a value of 90 degrees works best. The Character Controller will not be able to climb up walls due to the capsule shape.

Don't get stuck

The **Skin Width** is one of the most critical properties to get right when tuning your Character Controller. If your character gets stuck it is most likely because your **Skin Width** is too small. The **Skin Width** will let objects slightly penetrate the Controller but it removes jitter and prevents it from getting stuck.

It's good practice to keep your **Skin Width** at least greater than 0.01 and more than 10% of the **Radius**.

We recommend keeping **Min Move Distance** at 0.

See the Character Controller script reference here

You can download an example project showing pre-setup animated and moving character controllers from the Resources area on our website.

Hints

Try adjusting your **Skin Width** if you find your character getting stuck frequently.
The Character Controller can affect objects using physics if you write your own **scripts**.
The Character Controller can not be affected by objects through physics.
Note that changing Character Controller properties in the **inspector** will recreate the controller in the **scene**, so any existing Trigger contacts will get lost, and you will not get any OnTriggerEntered messages until the controller is moved again.
The Character Controller's capsule used in queries such as raycast might shrink by a small factor. Queries therefore could miss in some corner cases, even when they appear to hit the Character Controller's **gizmo**.

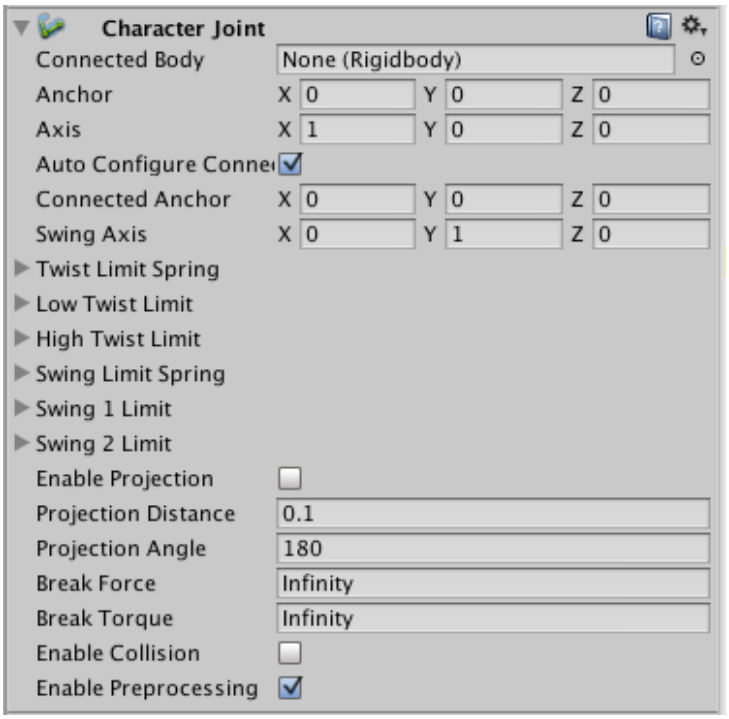
Character Joint

Leave feedback

SWITCH TO SCRIPTING

Character Joints are mainly used for Ragdoll effects. They are an extended ball-socket **joint** which allows you to limit the **joint** on each axis.

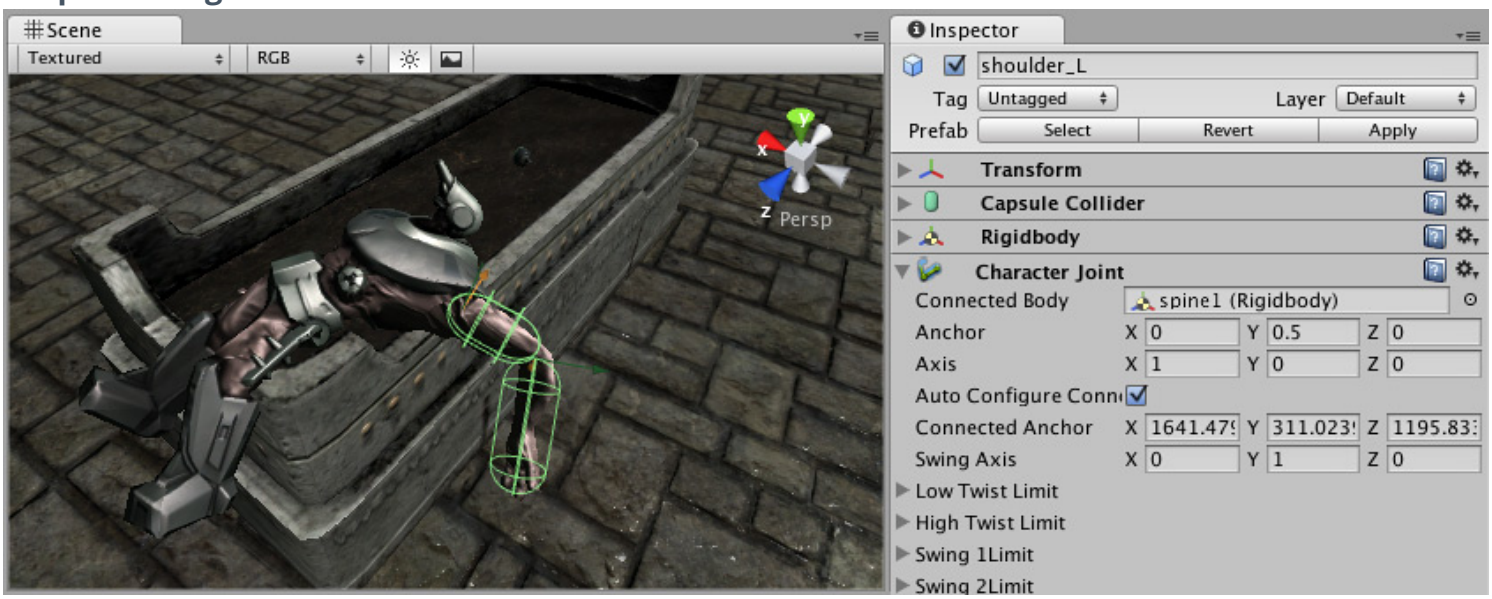
If you just want to set up a ragdoll read about Ragdoll Wizard.



Properties

Property:	*Function:*
Connected Body	Optional reference to the **Rigidbody** that the joint is dependent upon. If not set, the joint connects to the world.
Anchor	The point in the **GameObject's** local space where the joint rotates around.
Axis	The twist axes. Visualized with the orange **gizmo** cone.
Auto Configure Connected Anchor	If this is enabled, then the Connected Anchor position will be calculated automatically to match the global position of the anchor property. This is the default behavior. If this is disabled, you can configure the position of the connected anchor manually.
Connected Anchor	Manual configuration of the connected anchor position.
Swing Axis	The swing axis. Visualized with the green gizmo cone.
Low Twist Limit	The lower limit of the joint. See below.
High Twist Limit	The higher limit of the joint. See below.
Swing 1 Limit	Limits the rotation around one element of the defined **Swing Axis** (visualized with the green axis on the gizmo). See below.
Swing 2 Limit	Limits movement around one element of the defined **Swing Axis**. See below.

Property:	*Function:*
Break Force	The force that needs to be applied for this joint to break.
Break Torque	The torque that needs to be applied for this joint to break.
Enable Collision	When checked, this enables **collisions** between bodies connected with a joint.
Enable Preprocessing	Disabling preprocessing helps to stabilize impossible-to-fulfil configurations.



The Character Joint on a Ragdoll

Details

Character joints give you a lot of possibilities for constraining motion like with a universal joint.

The twist axis (visualized with the orange access on the gizmo) gives you most control over the limits as you can specify a lower and upper limit in degrees (the limit angle is measured relative to the starting position). A value of –30 in **Low Twist Limit**->**Limit** and 60 in **High Twist Limit**->**Limit** limits the rotation around the twist axis (orange gizmo) between –30 and 60 degrees.

The **Swing 1 Limit** limits the rotation around the swing axis (visualized with the green axis on the gizmo). The limit angle is symmetric. Thus a value of 30 will limit the rotation between –30 and 30.

The **Swing 2 Limit** axis isn't visualized on the gizmo but the axis is orthogonal to the two other axes (that is the twist axis visualised in orange on the gizmo and the **Swing 1 Limit** visualised in green on the gizmo). The angle is symmetric, thus a value of 40 will limit the rotation around that axis between –40 and 40 degrees.

For each of the limits the following values can be set:

Property:	*Function:*
Bounciness	A value of 0 will not bounce. A value of 1 will bounce without any loss of energy.
Spring	The spring force used to keep the two objects together.
Damper	The damper force used to dampen the spring force.
Contact Distance	Within the contact distance from the limit contacts will persist in order to avoid jitter.

Breaking joints

You can use the **Break Force** and **Break Torque** properties to set limits for the joint's strength. If these are less than infinity, and a force/torque greater than these limits are applied to the object, its **Fixed Joint** will be destroyed and will no longer be confined by its restraints.

Hints

You do not need to assign a **Connected Body** to your joint for it to work.
Character Joints require your object to have a Rigidbody attached.
For Character Joints made with the Ragdoll wizard, take a note that the setup is made such that the joint's Twist axis corresponds with the limb's largest swing axis, the joint's Swing 1 axis corresponds with limb's smaller swing axis and joint's Swing 2 is for twisting the limb. This naming scheme is for legacy reasons.

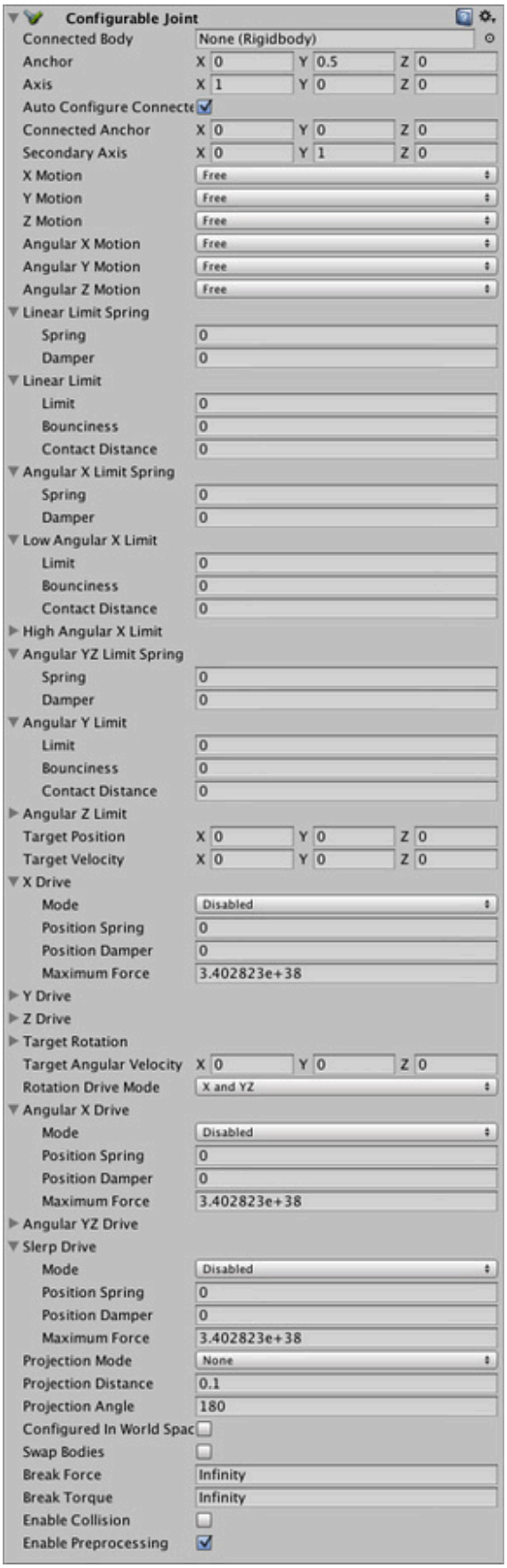
Configurable Joint

Leave feedback

SWITCH TO SCRIPTING

Configurable Joints are extremely customisable since they incorporate all the functionality of the other joint types. You can use them to create anything from adapted versions of the existing **joints** to highly specialised **joints** of your own design.

Properties



Property:	*Function:*
Connected Body	The other **Rigidbody** object to which the joint is connected. You can set this to *None* to indicate that the joint is attached to a fixed position in space rather than another **Rigidbody**.

Property:	*Function:*
Anchor	The point where the center of the joint is defined. All physics-based simulation will use this point as the center in calculations
Axis	The local axis that will define the object's natural rotation based on physics simulation
Auto Configure Connected Anchor	If this is enabled, then the Connected Anchor position will be calculated automatically to match the global position of the anchor property. This is the default behavior. If this is disabled, you can configure the position of the connected anchor manually.
Connected Anchor	Manual configuration of the connected anchor position.
Secondary Axis	Together, **Axis** and **Secondary Axis** define the local coordinate system of the joint. The third axis is set to be orthogonal to the other two.
X, Y, Z Motion	Allow movement along the X, Y or Z axes to be *Free*, completely *Locked*, or *Limited* according to the limit properties described below.
Angular X, Y, Z Motion	Allow rotation around the X, Y or Z axes to be *Free*, completely *Locked*, or *Limited* according to the limit properties described below.
Linear Limit Spring	A spring force applied to pull the object back when it goes past the limit position.
Spring	The spring force. If this value is set to zero then the limit will be impassable; a value other than zero will make the limit elastic.
Damper	The reduction of the spring force in proportion to the speed of the joint's movement. Setting a value above zero allows the joint to "dampen" oscillations which would otherwise carry on indefinitely.
Linear Limit	Limit on the joint's linear movement (ie, movement over distance rather than rotation), specified as a distance from the joint's origin.
Limit	The distance in world units from the origin to the limit.
Bounciness	Bounce force applied to the object to push it back when it reaches the limit distance.
Contact Distance	The minimum distance tolerance (between the joint position and the limit) at which the limit will be enforced. A high tolerance makes the limit less likely to be violated when the object is moving fast. However, this will also require the limit to be taken into account by the physics simulation more often and this will tend to reduce performance slightly.
Angular X Limit Spring	A spring torque applied to rotate the object back when it goes past the limit angle of the joint.
Spring	The spring torque. If this value is set to zero then the limit will be impassable; a value other than zero will make the limit elastic.
Damper	The reduction of the spring torque in proportion to the speed of the joint's rotation. Setting a value above zero allows the joint to "dampen" oscillations which would otherwise carry on indefinitely.
Low Angular X Limit	Lower limit on the joint's rotation around the X axis, specified as a angle from the joint's original rotation.
Limit	The limit angle.
Bounciness	Bounce torque applied to the object when its rotation reaches the limit angle.

Property:	*Function:*
Contact Distance	The minimum angular tolerance (between the joint angle and the limit) at which the limit will be enforced. A high tolerance makes the limit less likely to be violated when the object is moving fast. However, this will also require the limit to be taken into account by the physics simulation more often and this will tend to reduce performance slightly.
High Angular XLimit	This is similar to the *Low Angular X Limit* property described above but it determines the upper angular limit of the joint's rotation rather than the lower limit.
Angular YZ Limit Spring	This is similar to the *Angular X Limit Spring* described above but applies to rotation around both the Y and Z axes.
Angular Y Limit	Analogous to the *Angular X Limit* properties described above but applies to the Y axis and regards both the upper and lower angular limits as being the same.
Angular Z Limit	Analogous to the *Angular X Limit* properties described above but applies to the Z axis and regards both the upper and lower angular limits as being the same.
Target Position	The target position that the joint's drive force should move it to.
Target Velocity	The desired velocity with which the joint should move to the *Target Position* under the drive force.
XDrive	The drive force that moves the joint linearly along its local X axis.
Mode	The mode determines whether the joint should move to reach a specified **Position**, a specified **Velocity** or both.
Position Spring	The spring force that moves the joint towards its target position. This is only used when the drive mode is set to *Position* or *Position and Velocity*.
Position Damper	The reduction of the spring force in proportion to the speed of the joint's movement. Setting a value above zero allows the joint to "dampen" oscillations which would otherwise carry on indefinitely. This is only used when the drive mode is set to *Position* or *Position and Velocity*.
Maximum Force	The force used to accelerate the joint toward its target velocity. This is only used when the drive mode is set to *Velocity* or *Position and Velocity*.
YDrive	This is analogous to the *X Drive* described above but applies to the joint's Y axis.
ZDrive	This is analogous to the *X Drive* described above but applies to the joint's Z axis.
Target Rotation	The orientation that the joint's rotational drive should rotate towards, specified as a quaternion.
Target Angular Velocity	The angular velocity that the joint's rotational drive should aim to achieve. This is specified as a vector whose length specifies the rotational speed and whose direction defines the axis of rotation.
Rotation Drive Mode	The way in which the drive force will be applied to the object to rotate it to the target orientation. If the mode is set to *X and YZ*, the torque will be applied around these axes as specified by the *Angular X/YZ Drive* properties described below. If **Slerp** mode is used then the *Slerp Drive* properties will determine the drive torque.
Angular X Drive	This specifies how the joint will be rotated by the drive torque around its local X axis. It is used only if the *Rotation Drive Mode* property described above is set to *X & YZ*.
Mode	The mode determines whether the joint should move to reach a specified angular **Position**, a specified angular **Velocity** or both.
Position Spring	The spring torque that rotates the joint towards its target position. This is only used when the drive mode is set to *Position* or *Position and Velocity*.

Property:	*Function:*
Position Damper	The reduction of the spring torque in proportion to the speed of the joint's movement. Setting a value above zero allows the joint to "dampen" oscillations which would otherwise carry on indefinitely. This is only used when the drive mode is set to *Position* or *Position and Velocity*.
Maximum Force	The torque used to accelerate the joint toward its target velocity. This is only used when the drive mode is set to *Velocity* or *Position and Velocity*.
Angular YZDrive	This is analogous to the *Angular X Drive* described above but applies to both the joint's Y and Z axes.
Slerp Drive	This specifies how the joint will be rotated by the drive torque around all local axes. It is used only if the *Rotation Drive Mode* property described above is set to *Slerp*.
Mode	The mode determines whether the joint should move to reach a specified angular **Position**, a specified angular **Velocity** or both.
Position Spring	The spring torque that rotates the joint towards its target position. This is only used when the drive mode is set to *Position* or *Position and Velocity*.
Position Damper	The reduction of the spring torque in proportion to the speed of the joint's movement. Setting a value above zero allows the joint to "dampen" oscillations which would otherwise carry on indefinitely. This is only used when the drive mode is set to *Position* or *Position and Velocity*.
Maximum Force	The torque used to accelerate the joint toward its target velocity. This is only used when the drive mode is set to *Velocity* or *Position and Velocity*.
Projection Mode	This defines how the joint will be snapped back to its constraints when it unexpectedly moves beyond them (due to the physics engine being unable to reconcile the current combination of forces within the simulation). The options are *None* and *Position and Rotation*.
Projection Distance	The distance the joint must move beyond its constraints before the **physics engine** will attempt to snap it back to an acceptable position.
Projection Angle	The angle the joint must rotate beyond its constraints before the physics engine will attempt to snap it back to an acceptable position.
Configured in World Space	Should the values set by the various target and drive properties be calculated in world space instead of the object's local space?
Swap Bodies	If enabled, this will make the joint behave as though the **component** were attached to the connected Rigidbody (ie, the other end of the joint).
Break Force	If the joint is pushed beyond its constraints by a force larger than this value then the joint will be permanently "broken" and deleted.
Break Torque	If the joint is rotated beyond its constraints by a torque larger than this value then the joint will be permanently "broken" and deleted.
Enable Collision	Should the object with the joint be able to collide with the connected object (as opposed to just passing through each other)?
Enable Preprocessing	If preprocessing is disabled then certain "impossible" configurations of the joint will be kept more stable rather than drifting wildly out of control.

Details

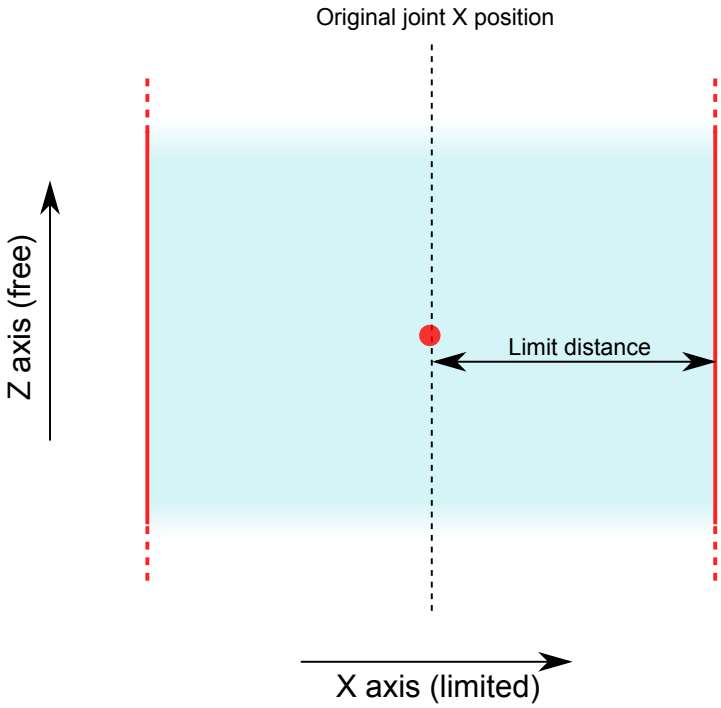
Like the other joints, the Configurable Joint allows you to restrict the movement of an object but also to drive it to a target velocity or position using forces. However, there are many configuration options available and they can be

quite subtle when used in combination; you may need to experiment with different options to get the joint to behave exactly the way you want.

Constraining Movement

You can constrain both translational movement and rotation on each axis independently using the *X, Y, Z Motion* and *X, Y, Z Rotation* properties. If *Configured In World Space* is enabled then movements will be constrained to the world axes rather than the object's local axes. Each of these properties can be set to *Locked*, *Limited* or *Free*:

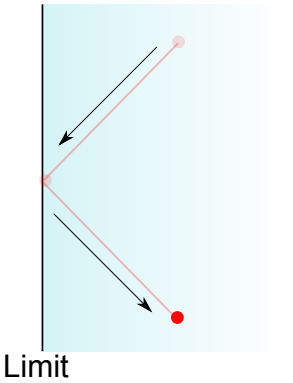
A **Locked** axis will allow no movement at all. For example, an object locked in the world Y axis cannot move up or down.
A **Limited** axis allows free movement between predefined limits, as explained below. For example, a gun turret might be given a restricted arc of fire by limiting its Y rotation to a specific angular range.
A **Free** axis allows any movement.
You can limit translational movement using the *Linear Limit* property, which defines the maximum distance the joint can move from its point of origin. (measured along each axis separately). For example, you could constrain the puck for an air hockey table by locking the joint in the Y axis (in world space), leaving it free in the Z axis and setting the limit for the X axis to fit the width of the table; the puck would then be constrained to stay within the playing area.



You can also limit rotation using the *Angular Limit* properties. Unlike the linear limit, these allow you to specify different limit values for each axis. Additionally, you can also define separate upper and lower limits on the angle of rotation for the X axis (the other two axes use the same angle either side of the original rotation). For example, you could construct a "teeter table" using a flat plane with a joint constrained to allow slight tilting in the X and Z directions while leaving the Y rotation locked.

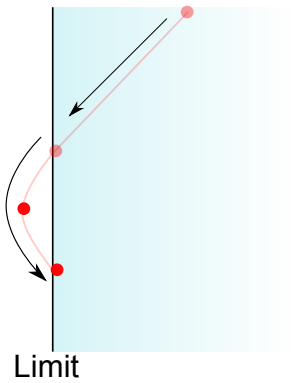
Bounciness and Springs

By default, a joint simply stops moving when it runs into its limit. However, an inelastic **collision** like this is rare in the real world and so it is useful to add some feeling of bounce to a constrained joint. You can use the *Bounciness* property of the linear and angular limits to make the constrained object bounce back after it hits its limit. Most collisions will look more natural with a small amount of bounciness but you can also set this property higher to simulate unusually bouncy boundaries like, say, the cushions of a pool table.



Bouncy joint does not cross the limit

The joint limits can be softened further using the spring properties: *Linear Limit Spring* for translation and *Angular X/YZ Limit Spring* for rotation. If you set the *Spring* property to a value above zero, the joint will not abruptly stop moving when it hits a limit but will be drawn back to the limit position by a spring force (the strength of the force is determined by the *Spring* value). By default, the spring is perfectly elastic and will tend to catapult the joint back in the direction opposite to the collision. However, you can use the *Damper* property to reduce the elasticity and return the joint to the limit more gently. For example, you might use a **spring joint** to create a lever that can be pulled to the left or right but then springs back to an upright position. If the springs are perfectly elastic then the lever will tend to oscillate back and forth around the centre point after it is released. However, if you add enough damping then the spring will rapidly settle down to the neutral position.



Spring joint crosses the limit but is pulled back to it

Drive forces

Not only can a joint react to the movements of the attached object, but it can also actively apply *drive forces* to set the object in motion. Some joints simple need to keep the object moving at a constant speed as with, say, a rotary motor turning a fan blade. You can set your desired velocity for such joints using the *Target Velocity* and *Target Angluar Velocity* properties. You might also require joints that move their object towards a particular position in space (or a particular orientation); you can set these using the *Target Position* and *Target Rotation* properties. For example, you could implement a forklift by mounting the forks on a configurable joint and then setting the target height to raise them from a script.

With the target set, the *X, Y, Z Drive* and *Angular X/YZ Drive* (or alternatively *Slerp Drive*) properties then specify the force used to push the joint toward it. The Drives' *Mode* property selects whether the joint should seek a target position, velocity or both. The *Position Spring* and *Position Damper* work in the same way as for the joint limits when seeking a target position. In velocity mode, the spring force is dependent on the "distance" between the current velocity and the target velocity; the damper helps the velocity to settle at the chosen value rather than oscillating endlessly around it. The *Maximum Force* property is a final refinement that prevents the force applied by the spring

from exceeding a limit value regardless of how far the joint is from its target. This prevents the circumstance where a joint stretched far from its target rapidly snaps the object back in an uncontrolled way.

Note that with all the drive forces (except for *Slerp Drive*, described below), the force is applied separately in each axis. So, for example, you could implement a spacecraft that has a high forward flying speed but a relatively low speed in sideways steering motion.

Slerp Drive

While the other drive modes apply forces in separate axes, *Slerp Drive* uses the Quaternion's spherical **interpolation** or "slerp" functionality to reorient the joint. Rather than isolating individual axes, the slerp process essentially finds the minimum total rotation that will take the object from the current orientation to the target and applies it on all axes as necessary. Slerp drive is slightly easier to configure and fine for most purposes but does not allow you to specify different drive forces for the X and Y/Z axes.

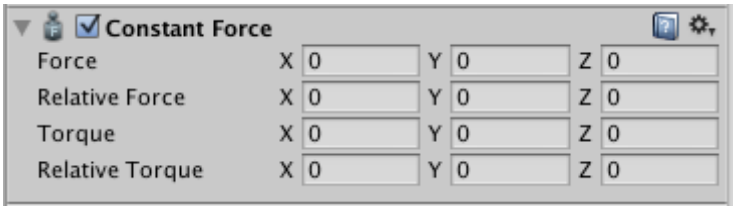
To enable Slerp drive, you should change the *Rotation Drive Mode* property from *X and YZ* to *Slerp*. Note that the modes are mutually exclusive; the joint will use either the *Angular X/YZ Drive* values or the *Slerp Drive* values but not both together.

Constant Force

Leave feedback

SWITCH TO SCRIPTING

Constant Force is a quick utility for adding constant forces to a **Rigidbody**. This works great for one shot objects like rockets, if you don't want it to start with a large velocity but instead accelerate.



Properties

Property:	*Function:*
Force	The vector of a force to be applied in world space.
Relative Force	The vector of a force to be applied in the object's local space.
Torque	The vector of a torque, applied in world space. The object will begin spinning *around* this vector. The longer the vector is, the faster the rotation.
Relative Torque	The vector of a torque, applied in local space. The object will begin spinning *around* this vector. The longer the vector is, the faster the rotation.

Details

To make a rocket that accelerates forward set the **Relative Force** to be along the positive z-axis. Then use the Rigidbody's **Drag** property to make it not exceed some maximum velocity (the higher the drag the lower the maximum velocity will be). In the Rigidbody, also make sure to turn off gravity so that the rocket will always stay on its path.

Hints

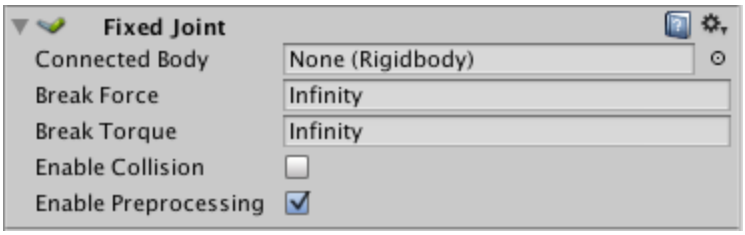
To make an object flow upwards, add a Constant Force with the **Force** property having a positive Y value.
To make an object fly forwards, add a Constant Force with the **Relative Force** property having a positive Z value.

Fixed Joint

Leave feedback

SWITCH TO SCRIPTING

Fixed Joints restricts an object's movement to be dependent upon another object. This is somewhat similar to **Parenting** but is implemented through physics rather than **Transform** hierarchy. The best scenarios for using them are when you have objects that you want to easily break apart from each other, or connect two object's movement without parenting.



Properties

Property:	*Function:*
Connected Body	Optional reference to the Rigidbody that the **joint** is dependent upon. If not set, the **joint** connects to the world.
Break Force	The force that needs to be applied for this joint to break.
Break Torque	The torque that needs to be applied for this joint to break.
Enable Collision	When checked, this enables **collisions** between bodies connected with a joint.
Enable Preprocessing	Disabling preprocessing helps to stabilize impossible-to-fulfil configurations.

Details

There may be scenarios in your game where you want objects to stick together permanently or temporarily. Fixed Joints may be a good **Component** to use for these scenarios, since you will not have to script a change in your object's hierarchy to achieve the desired effect. The trade-off is that you must use **Rigidbodies** for any objects that use a Fixed Joint.

For example, if you want to use a "sticky grenade", you can write a script that will detect collision with another Rigidbody (like an enemy), and then create a Fixed Joint that will attach itself to that Rigidbody. Then as the enemy moves around, the joint will keep the grenade stuck to them.

Breaking joints

You can use the **Break Force** and **Break Torque** properties to set limits for the joint's strength. If these are less than infinity, and a force/torque greater than these limits are applied to the object, its Fixed Joint will be destroyed and will no longer be confined by its restraints.

Hints

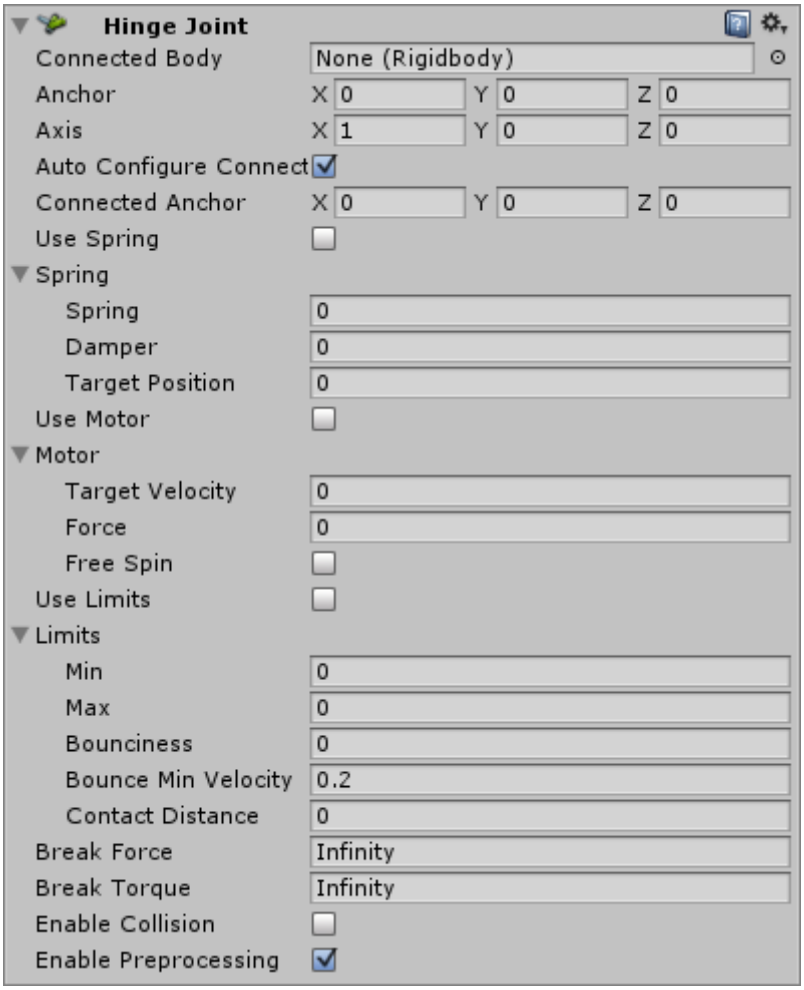
You do not need to assign a **Connected Body** to your joint for it to work.
Fixed Joints require a Rigidbody.

Hinge Joint

Leave feedback

SWITCH TO SCRIPTING

The **Hinge Joint** groups together two Rigidbodies, constraining them to move like they are connected by a hinge. It is perfect for doors, but can also be used to model chains, pendulums, **etc**.



Properties

Property:	*Function:*
Connected Body	Optional reference to the Rigidbody that the **joint** is dependent upon. If not set, the **joint** connects to the world.
Anchor	The position of the axis around which the body swings. The position is defined in local space.
Axis	The direction of the axis around which the body swings. The direction is defined in local space.
Auto Configure Connected Anchor	If this is enabled, then the Connected Anchor position will be calculated automatically to match the global position of the anchor property. This is the default behavior. If this is disabled, you can configure the position of the connected anchor manually.
Connected Anchor	Manual configuration of the connected anchor position.

Property:	*Function:*
Use Spring	Spring makes the Rigidbody reach for a specific angle compared to its connected body.
Spring	Properties of the Spring that are used if **Use Spring** is enabled.
Spring	The force the object asserts to move into the position.
Damper	The higher this value, the more the object will slow down.
Target Position	Target angle of the spring. The spring pulls towards this angle measured in degrees.
Use Motor	The motor makes the object spin around.
Motor	Properties of the Motor that are used if **Use Motor** is enabled.
Target Velocity	The speed the object tries to attain.
Force	The force applied in order to attain the speed.
Free Spin	If enabled, the motor is never used to brake the spinning, only accelerate it.
Use Limits	If enabled, the angle of the hinge will be restricted within the **Min** & **Max** values.
Limits __ |Properties of the Limits that are used if Use Limits__ is enabled.	
Min	The lowest angle the rotation can go.
Max	The highest angle the rotation can go.
Bounciness	How much the object bounces when it hits the minimum or maximum stop limit.
Contact Distance	Within the contact distance from the limit contacts will persist in order to avoid jitter.
Break Force	The force that needs to be applied for this joint to break.
Break Torque	The torque that needs to be applied for this joint to break.
Enable Collision	When checked, this enables **collisions** between bodies connected with a joint.
Enable Preprocessing	Disabling preprocessing helps to stabilize impossible-to-fulfil configurations.

Details

A single Hinge Joint should be applied to a **GameObject**. The hinge will rotate at the point specified by the **Anchor** property, moving around the specified **Axis** property. You **do not** need to assign a GameObject to the joint's **Connected Body** property. You should only assign a GameObject to the **Connected Body** property if you want the joint's **Transform** to be dependent on the attached object's Transform.

Think about how the hinge of a door works. The **Axis** in this case is up, positive along the Y axis. The **Anchor** is placed somewhere at the intersection between door and wall. You would not need to assign the wall to the **Connected Body**, because the joint will be connected to the world by default.

Now think about a doggy door hinge. The doggy door's **Axis** would be sideways, positive along the relative X axis. The main door should be assigned as the **Connected Body**, so the doggy door's hinge is dependent on the main door's Rigidbody.

Chains

Multiple Hinge Joints can also be strung together to create a chain. Add a joint to each link in the chain, and attach the next link as the **Connected Body**.

Hints

You do not need to assign a **Connected Body** to your joint for it to work.
Use **Break Force** in order to make dynamic damage systems. You can use this to allow the player to damage the environment (for example, break a door off its hinges by blasting it with a rocket launcher or running into it with a car).
The **Spring**, **Motor**, and **Limits** properties allow you to fine-tune your joint's behaviors.
Use of **Spring**, **Motor** are intended to be mutually exclusive. Using both at the same time leads to unpredictable results.

Mesh Collider

Leave feedback

SWITCH TO SCRIPTING

The **Mesh Collider** takes a Mesh Asset and builds its **Collider** based on that Mesh. It is far more accurate for **collision detection** than using primitives for complicated Meshes. Mesh Colliders that are marked as **Convex** can collide with other Mesh Colliders.

Properties

Property	*Function*
Convex	Tick the checkbox to enable **Convex**. If enabled, this Mesh Collider collides with other Mesh Colliders. **Convex** Mesh Colliders are limited to 255 triangles.
Is Trigger	If enabled, Unity uses this Collider for triggering events, and the **physics engine** ignores it.
Cooking Options	Enable or disable the Mesh cooking options that affect how the physics engine processes Meshes.
None	Disable all of the **Cooking Options** listed below.
Everything	Enable all of the **Cooking Options** listed below.
Inflate Convex Mesh	Allow the physics engine to increase the volume of the input Mesh, to generate a valid convex mesh.
Cook for Faster Simulation	Make the physics engine cook Meshes for faster simulation. When enabled, this runs some extra steps to guarantee the resulting Mesh is optimal for run-time performance. This affects the performance of the physics queries and contacts generation. When this setting is disabled, the physics engine uses a faster cooking time instead, and produces results as fast as possible. Consequently, the cooked Mesh Collider might not be optimal.
Enable Mesh Cleaning	Make the physics engine clean Meshes. When enabled, the cooking process tries to eliminate degenerate triangles of the Mesh, as well as other geometrical artifacts. This results in a Mesh that is better suited for use in **collision** detection and tends to produce more accurate hit points.
Weld Colocated Vertices	Make the physics engine remove equal vertices in the Meshes. When enabled, the physics engine combines the vertices that have the same position. This is important for the collision feedback that happens at run time.
Material	Reference to the Physics Material that determines how this Collider interacts with others.
Mesh	Reference to the Mesh to use for collisions.

Details

The Mesh Collider builds its collision representation from the Mesh attached to the **GameObject**, and reads the properties of the attached Transform to set its position and scale correctly. The benefit of this is that you can make the shape of the Collider exactly the same as the shape of the visible Mesh for the GameObject, resulting in more precise and authentic collisions. However, this precision comes with a higher processing overhead than collisions involving primitive colliders (such as Sphere, Box, and Capsule) and so it is best to use Mesh Colliders sparingly.

Faces in collision meshes are one-sided. This means objects can pass through them from one direction, but collide with them from the other.

Mesh cooking

Mesh cooking changes a normal Mesh into a Mesh that you can use in the physics engine. Cooking builds the spatial search structures for the physics queries, such as Physics.Raycast, as well as supporting structures for the contacts generation. Unity cooks all Meshes before using them in collision detection. This can happen at import time (**Import Settings > Model > Generate Colliders**) or at run time.

When generating Meshes at run time (for example, for procedural surfaces), it's useful to set the **Cooking Options** to produce results faster, and disable the additional data cleaning steps of cleaning. The downside is that you need to generate no degenerate triangles and no co-located vertices, but the cooking works faster.

If you disable **Enable Mesh Cleaning** or **Weld Colocated Vertices**, you need to ensure you aren't using data that those algorithms would otherwise filter. Make sure you don't have any co-located vertices if **Weld Colocated Vertices** is disabled, and when **Enable Mesh Cleaning** is enabled, make sure there are no tiny triangles whose area is close to zero, no thin triangles, and no huge triangles whose area is close to infinity.

Note: Setting **Cooking Options** to any other value than the default settings means the Mesh Collider must use a Mesh that has an isReadable value of true.

Limitations

There are some limitations when using the Mesh Collider:

Mesh Colliders that do not have **Convex** enabled are only supported on GameObjects without a **Rigidbody component**. To apply a Mesh Collider to a **Rigidbody component**, tick the **Convex** checkbox.

For a Mesh Collider to work properly, the Mesh must be read/write enabled in any of these circumstances:

The Mesh Collider's transform has negative scaling (for example, (–1, 1, 1)).
The Mesh Collider's transform is skewed or sheared (for example, when a rotated transform has a scaled parent transform).
The Mesh Collider's Cooking Options flags are set to any value other than the default.
Optimization tip: If a Mesh is used only by a Mesh Collider, you can disable **Normals** in **Import Settings**, because the physics system doesn't need them.

2018–06–07 Page amended with editorial review

2018–08–23 Page amended with editorial review

Mesh Cooking Options added in 2017.3

Updated functionality in 2018.1

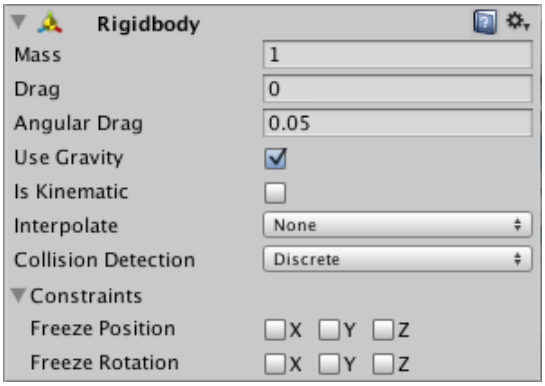
Updated limitations relating to read/write enabled setting in 2018.3

Rigidbody

Leave feedback

SWITCH TO SCRIPTING

Rigidbodies enable your **GameObjects** to act under the control of physics. The Rigidbody can receive forces and torque to make your objects move in a realistic way. Any GameObject must contain a Rigidbody to be influenced by gravity, act under added forces via scripting, or interact with other objects through the NVIDIA PhysX **physics engine**.



Properties

Property:	*Function:*
Mass	The mass of the object (in kilograms by default).
Drag	How much air resistance affects the object when moving from forces. 0 means no air resistance, and infinity makes the object stop moving immediately.
Angular Drag	How much air resistance affects the object when rotating from torque. 0 means no air resistance. Note that you cannot make the object stop rotating just by setting its Angular Drag to infinity.
Use Gravity	If enabled, the object is affected by gravity.
Is Kinematic	If enabled, the object will not be driven by the physics engine, and can only be manipulated by its **Transform**. This is useful for moving platforms or if you want to animate a Rigidbody that has a **HingeJoint** attached.
Interpolate	Try one of the options only if you are seeing jerkiness in your Rigidbody's movement.
- **None**	No **Interpolation** is applied.
- **Interpolate**	Transform is smoothed based on the Transform of the previous frame.
- **Extrapolate**	Transform is smoothed based on the estimated Transform of the next frame.
Collision Detection	Used to prevent fast moving objects from passing through other objects without detecting **collisions**.
- **Discrete**	Use Discrete collision detection against all other colliders in the **scene**. Other colliders will use Discrete collision detection when testing for collision against it. Used for normal collisions (This is the default value).
- **Continuous**	Use Discrete collision detection against dynamic colliders (with a rigidbody) and continuous collision detection against static MeshColliders (without a rigidbody). Rigidbodies set to Continuous Dynamic will use continuous collision detection when testing for collision against this rigidbody. Other rigidbodies will use Discrete Collision detection. Used for objects which the Continuous Dynamic detection needs to collide with. (This has a big impact on physics performance, leave it set to Discrete, if you don't have issues with collisions of fast objects)
- **Continuous Dynamic**	Use continuous collision detection against objects set to Continuous and Continuous Dynamic Collision. It will also use continuous collision detection against static MeshColliders (without a rigidbody). For all other colliders it uses Discrete collision detection. Used for fast moving objects.

Property:	*Function:*
Constraints	Restrictions on the Rigidbody's motion:-
- **Freeze Position**	Stops the Rigidbody moving in the world X, Y and Z axes selectively.
- **Freeze Rotation**	Stops the Rigidbody rotating around the local X, Y and Z axes selectively.

Details

Rigidbodies allow your GameObjects to act under control of the physics engine. This opens the gateway to behaviors such as realistic collisions and varied types of **joints**. Manipulating your GameObjects by adding forces to a Rigidbody creates a very different feel and look than adjusting the Transform **Component** directly. Generally, you shouldn't manipulate the Rigidbody and the Transform of the same GameObject - only one or the other.

The biggest difference between manipulating the Transform versus the Rigidbody is the use of forces. Rigidbodies can receive forces and torque, but Transforms cannot. Transforms can be translated and rotated, but this is not the same as using physics. You'll notice the distinct difference when you try it for yourself. Adding forces/torque to the Rigidbody will actually change the object's position and rotation of the **Transform component**. This is why you should only be using one or the other. Changing the Transform while using physics could cause problems with collisions and other calculations.

Rigidbodies must be explicitly added to your GameObject before they will be affected by the physics engine. You can add a Rigidbody to your selected object from **Components->Physics->Rigidbody** in the menu. Now your object is physics-ready; it will fall under gravity and can receive forces via scripting, but you may need to add a **Collider** or a Joint to get it to behave exactly how you want.

Parenting

When an object is under physics control, it moves semi-independently of the way its transform parents move. If you move any parents, they will pull the Rigidbody child along with them. However, the Rigidbodies will still fall down due to gravity and react to collision events.

Scripting

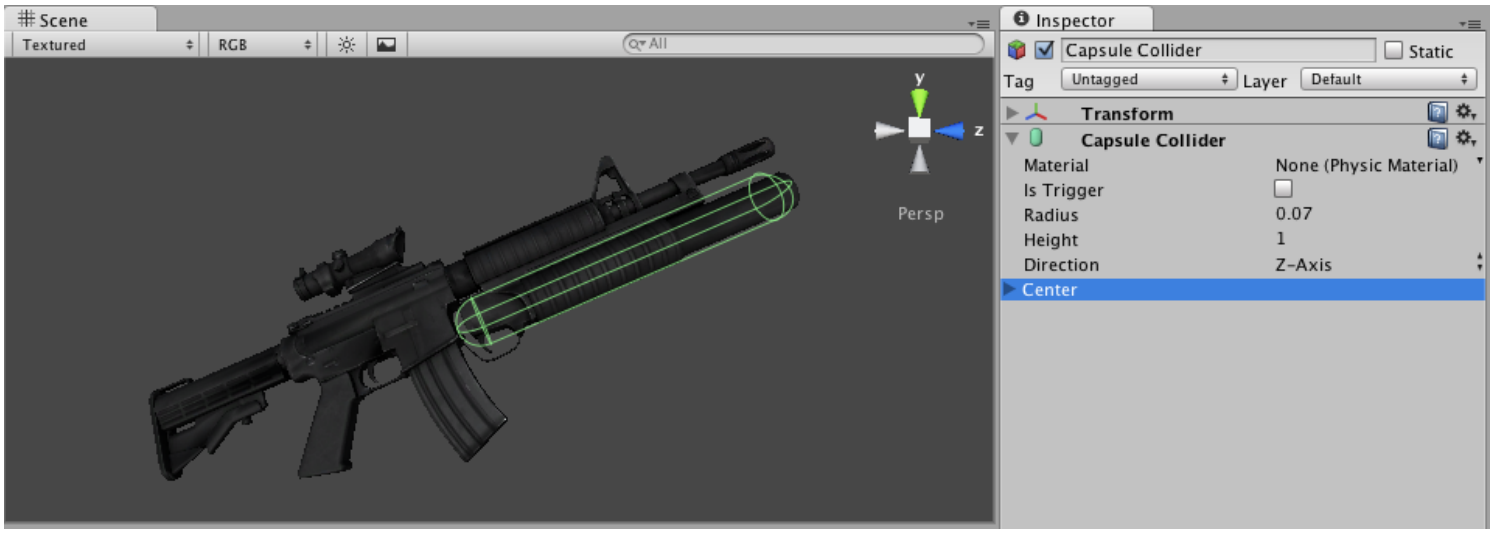
To control your Rigidbodies, you will primarily use **scripts** to add forces or torque. You do this by calling **AddForce()** and **AddTorque()** on the object's Rigidbody. Remember that you shouldn't be directly altering the object's Transform when you are using physics.

Animation

For some situations, mainly creating ragdoll effects, it is neccessary to switch control of the object between animations and physics. For this purpose Rigidbodies can be marked **isKinematic** . While the Rigidbody is marked **isKinematic**, it will not be affected by collisions, forces, or any other part of the physics system. This means that you will have to control the object by manipulating the Transform component directly. Kinematic Rigidbodies will affect other objects, but they themselves will not be affected by physics. For example, Joints which are attached to Kinematic objects will constrain any other Rigidbodies attached to them and Kinematic Rigidbodies will affect other Rigidbodies through collisions.

Colliders

Colliders are another kind of component that must be added alongside the Rigidbody in order to allow collisions to occur. If two Rigidbodies bump into each other, the physics engine will not calculate a collision unless both objects also have a Collider attached. Collider-less Rigidbodies will simply pass through each other during physics simulation.



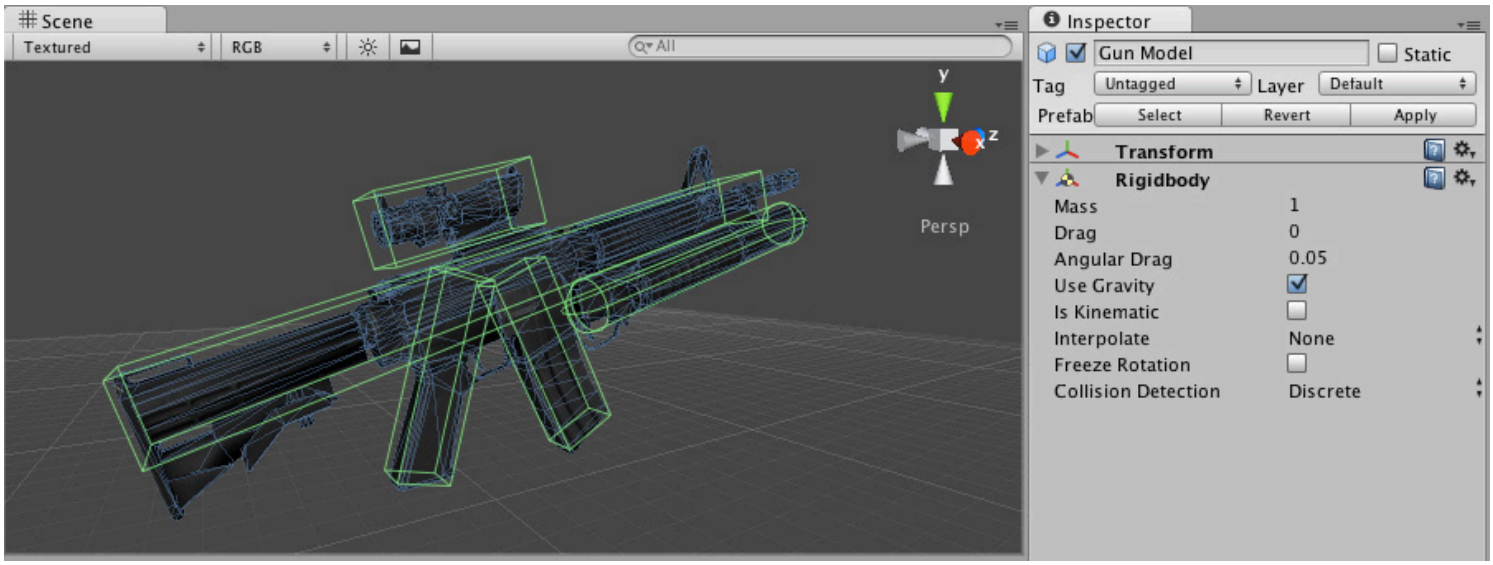
Colliders define the physical boundaries of a Rigidbody

Add a Collider with the **Component->Physics** menu. View the Component Reference page of any individual Collider for more specific information:

Box Collider - primitive shape of a cube
Sphere Collider - primitive shape of a sphere
Capsule Collider - primitive shape of a capsule
Mesh Collider - creates a collider from the object's **mesh**, cannot collide with another **Mesh** Collider
Wheel Collider - specifically for creating cars or other moving vehicles
Terrain Collider - handles collision with Unity's **terrain** system

Compound Colliders

Compound Colliders are combinations of primitive Colliders, collectively acting as a single Collider. They come in handy when you have a model that would be too complex or costly in terms of performance to simulate exactly, and want to simulate the collision of the shape in an optimal way using simple approximations. To create a Compound Collider, create child objects of your colliding object, then add a Collider component to each child object. This allows you to position, rotate, and scale each Collider easily and independently of one another. You can build your compound collider out of a number of primitive colliders and/or convex mesh colliders.



A real-world Compound Collider setup

In the above picture, the *Gun Model* GameObject has a Rigidbody attached, and multiple primitive Colliders as child GameObjects. When the Rigidbody parent is moved around by forces, the child Colliders move along with it. The primitive

Colliders will collide with the environment's Mesh Collider, and the parent Rigidbody will alter the way it moves based on forces being applied to it and how its child Colliders interact with other Colliders in the Scene.

Mesh Colliders can't normally collide with each other. If a Mesh Collider is marked as **Convex**, then it can collide with another Mesh Collider. The typical solution is to use primitive Colliders for any objects that move, and Mesh Colliders for static background objects.

Continuous Collision Detection

Continuous collision detection is a feature to prevent fast-moving colliders from passing each other. This may happen when using normal (**Discrete**) collision detection, when an object is one side of a collider in one frame, and already passed the collider in the next frame. To solve this, you can enable continuous collision detection on the rigidbody of the fast-moving object. Set the collision detection mode to **Continuous** to prevent the rigidbody from passing through any static (ie, non-rigidbody) MeshColliders. Set it to **Continuous Dynamic** to also prevent the rigidbody from passing through any other supported rigidbodies with collision detection mode set to **Continuous** or **Continuous Dynamic**. Continuous collision detection is supported for Box-, Sphere- and CapsuleColliders. Note that continuous collision detection is intended as a safety net to catch collisions in cases where objects would otherwise pass through each other, but will not deliver physically accurate collision results, so you might still consider decreasing the fixed Time step value in the TimeManager **inspector** to make the simulation more precise, if you run into problems with fast moving objects.

Use the right size

The size of the your GameObject's mesh is much more important than the mass of the Rigidbody. If you find that your Rigidbody is not behaving exactly how you expect - it moves slowly, floats, or doesn't collide correctly - consider adjusting the scale of your mesh **asset**. Unity's default unit scale is 1 unit = 1 meter, so the scale of your imported mesh is maintained, and applied to physics calculations. For example, a crumbling skyscraper is going to fall apart very differently than a tower made of toy blocks, so objects of different sizes should be modeled to accurate scale.

If you are modeling a human make sure the model is around 2 meters tall in Unity. To check if your object has the right size compare it to the default cube. You can create a cube using **GameObject > 3D Object > Cube**. The cube's height will be exactly 1 meter, so your human should be twice as tall.

If you aren't able to adjust the mesh itself, you can change the uniform scale of a particular mesh asset by selecting it in **Project View** and choosing **Assets->Import Settings...** from the menu. Here, you can change the scale and re-import your mesh.

If your game requires that your GameObject needs to be instantiated at different scales, it is okay to adjust the values of your Transform's scale axes. The downside is that the physics simulation must do more work at the time the object is instantiated, and could cause a performance drop in your game. This isn't a terrible loss, but it is not as efficient as finalizing your scale with the other two options. Also keep in mind that non-uniform scales can create undesirable behaviors when Parenting is used. For these reasons it is always optimal to create your object at the correct scale in your modeling application.

Hints

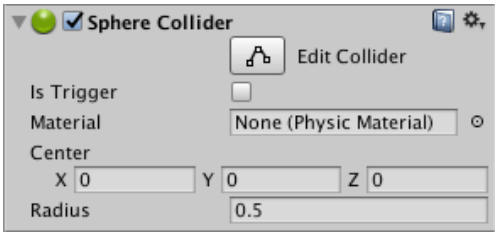
The relative **Mass** of two Rigidbodies determines how they react when they collide with each other.
Making one Rigidbody have greater **Mass** than another does not make it fall faster in free fall. Use **Drag** for that.
A low **Drag** value makes an object seem heavy. A high one makes it seem light. Typical values for **Drag** are between .001 (solid block of metal) and 10 (feather).
If you are directly manipulating the Transform component of your object but still want physics, attach a Rigidbody and make it Kinematic.
If you are moving a GameObject through its Transform component but you want to receive Collision/Trigger messages, you must attach a Rigidbody to the object that is moving.
You cannot make an object stop rotating just by setting its Angular Drag to infinity.

Sphere Collider

Leave feedback

SWITCH TO SCRIPTING

The **Sphere Collider** is a basic sphere-shaped **collision** primitive.



Properties

Property:	*Function:*
Is Trigger	If enabled, this **Collider** is used for triggering events, and is ignored by the **physics engine**.
Material	Reference to the Physics Material that determines how this Collider interacts with others.
Center	The position of the Collider in the object's local space.
Radius	The size of the Collider.

Details

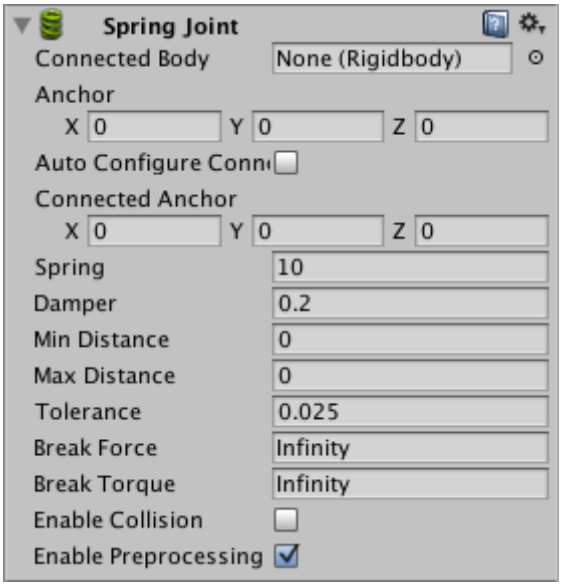
The collider can be resized via the **Radius** property but cannot be scaled along the three axes independently (ie, you can't flatten the sphere into an ellipse). As well as the obvious use for spherical objects like tennis balls, **etc**, the sphere also works well for falling boulders and other objects that need to roll and tumble.

Spring Joint

Leave feedback

SWITCH TO SCRIPTING

The **Spring Joint** joins two Rigidbodies together but allows the distance between them to change as though they were connected by a spring.



Properties

Property:	*Function:*
Connected Body	The Rigidbody object that the object with the spring **joint** is connected to. If no object is assigned then the spring will be connected to a fixed point in space.
Anchor	The point in the object's local space at which the joint is attached.
Auto Configure Connected Anchor	Should Unity calculate the position of the connected anchor point automatically?
Connected Anchor	The point in the connected object's local space at which the joint is attached.
Spring	Strength of the spring.
Damper	Amount that the spring is reduced when active.
Min Distance	Lower limit of the distance range over which the spring will not apply any force.
Max Distance	Upper limit of the distance range over which the spring will not apply any force.
Tolerance	Changes error tolerance. Allows the spring to have a different rest length.
Break Force	The force that needs to be applied for this joint to break.
Break Torque	The torque that needs to be applied for this joint to break.
Enable Collision	Should the two connected objects register **collisions** with each other?
Enable Preprocessing	Disabling preprocessing helps to stabilize impossible-to-fulfil configurations.

Details

The spring acts like a piece of elastic that tries to pull the two anchor points together to the exact same position. The strength of the pull is proportional to the current distance between the points with the force per unit of distance set by the *Spring* property. To prevent the spring from oscillating endlessly you can set a *Damper* value that reduces the spring force in proportion to the relative speed between the two objects. The higher the value, the more quickly the oscillation will die down.

You can set the anchor points manually but if you enable *Auto Configure Connected Anchor*, Unity will set the connected anchor so as to maintain the initial distance between them (ie, the distance you set in the scene view while positioning the objects).

The *Min Distance* and *Max Distance* values allow you to set a distance range over which the spring will not apply any force. You could use this, for example, to allow the objects a small amount of independent movement but then pull them together when the distance between them gets too great.

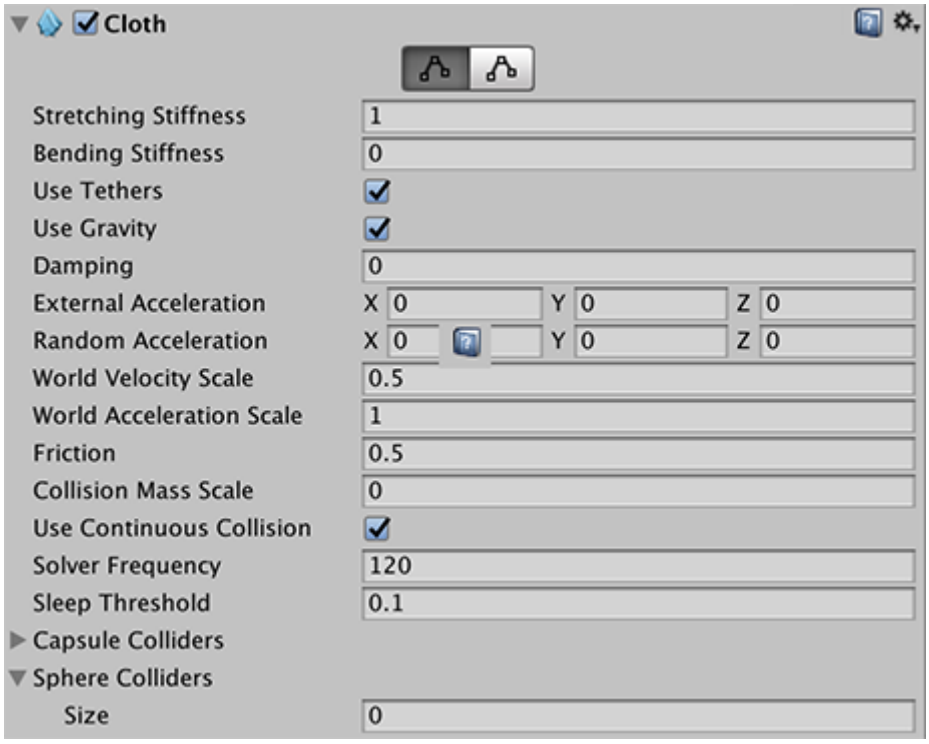
Cloth

Leave feedback

SWITCH TO SCRIPTING

The Cloth **component** works with the Skinned **Mesh** Renderer to provide a physics-based solution for simulating fabrics. It is specifically designed for character clothing, and only works with skinned meshes. If you add a Cloth **component** to a non-skinned **Mesh**, Unity removes the non-skinned **Mesh** and adds a skinned **Mesh**.

To attach a Cloth component to a skinned Mesh, select the GameObject in the Editor, click the **Add Component** button in the Inspector window, and select **Physics** > **Cloth**. The component appears in the Inspector.



Properties

Property:	*Function:*
Stretching Stiffness	Stretching stiffness of the cloth.
Bending Stiffness	Bending stiffness of the cloth.
Use Tethers	Apply constraints that help to prevent the moving cloth particles from going too far away from the fixed ones. This helps to reduce excess stretchiness.
Use Gravity	Should gravitational acceleration be applied to the cloth?
Damping	Motion damping coefficient.
External Acceleration	A constant, external acceleration applied to the cloth.
Random Acceleration	A random, external acceleration applied to the cloth.
World Velocity Scale	How much world-space movement of the character will affect cloth vertices.

Property:	*Function:*
World Acceleration Scale	How much world-space acceleration of the character will affect cloth vertices.
Friction	The friction of the cloth when colliding with the character.
Collision Mass Scale	How much to increase mass of colliding particles.
Use Continuous Collision	Enable continuous **collision** to improve **collision** stability.
Use Virtual Particles	Add one virtual particle per triangle to improve collision stability.
Solver Frequency	Number of solver iterations per second.
Sleep Threshold	Cloth's sleep threshold.
Capsule Colliders	An array of CapsuleColliders which this Cloth instance should collide with.
Sphere Colliders	An array of ClothSphereColliderPairs which this Cloth instance should collide with.

Details

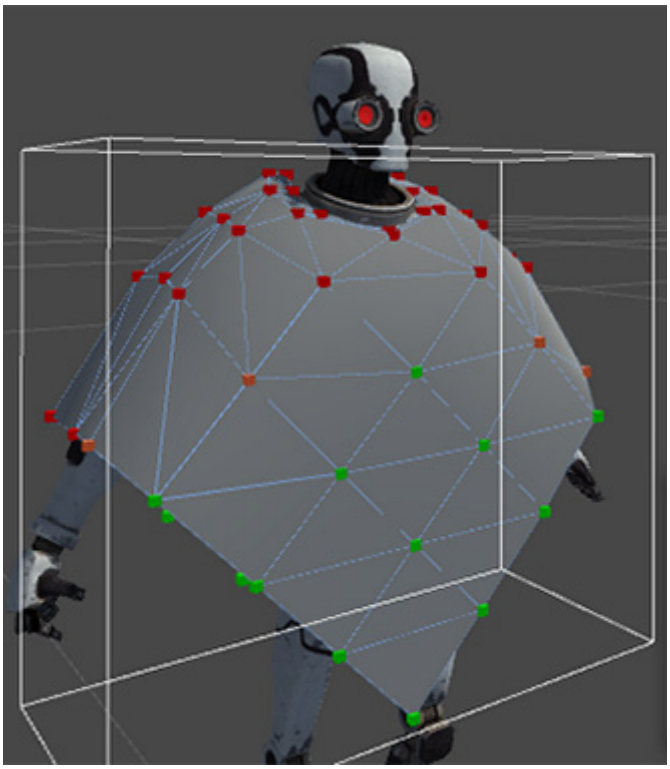
Cloth does not react to all **colliders** in a **scene**, nor does it apply forces back to the world. When it has been added the Cloth component will not react to or influence any other bodies at all. Thus Cloth and the world do not recognise or see each other until you manually add **colliders** from the world to the Cloth component. Even after that, the simulation is still one-way: cloth reacts to those bodies but doesn't apply forces back.

Additionally, you can only use three types of colliders with cloth: a sphere, a capsule, and conical capsule colliders, constructed using two sphere colliders. These restrictions all exist to help boost performance.

Edit Constraints Tool

Select **Edit** > **Constraints** to edit the constraints applied to each of the vertices in the cloth mesh. All vertices have a color based on the current visualization mode, to display the difference between their respective values. You can author Cloth constraints by painting them onto the cloth with a brush.

Property:	*Function:*
Visualization	Changes the visual appearance of the tool in the **Scene view** between Max Distance and Surface Penetration Values. A toggle for Manipulate Backfaces is also available.
Max Distance	The maximum distance a cloth particle can travel from its vertex position.
Surface Penetration	How deep the cloth particle can penetrate the mesh.
Brush Radius	Sets the radius of a brush that enables you to paint constraints onto a cloth.



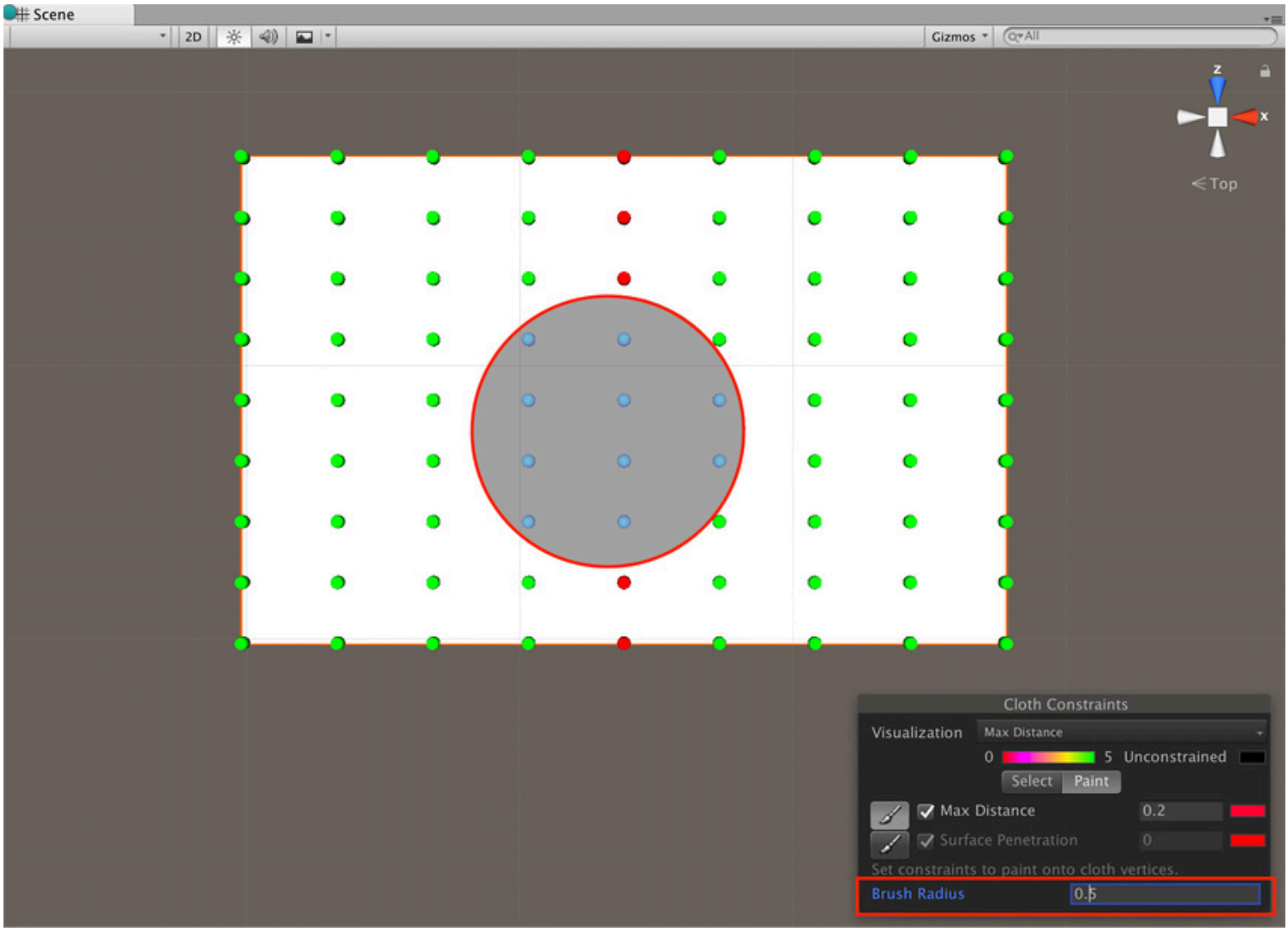
The Cloth Constraints Tool being used on a Skinned Mesh Renderer.

There are two modes for changing the values for each vertex:

Use **Select** mode to select a group of vertices. To do this, use the mouse cursor to draw a selection box or click on vertices one at a time. You can can then enable **Max Distance**, **Surface Penetration**, or both, and set a value.

Use **Paint** mode to directly adjust each individual vertex. To do this, click the vertex you want to adjust. You can can then enable **Max Distance**, **Surface Penetration**, or both, and set a value.

In both modes, the visual representation in the Scene view automatically updates when you assign values to **Max Distance** and **Surface Penetration**.



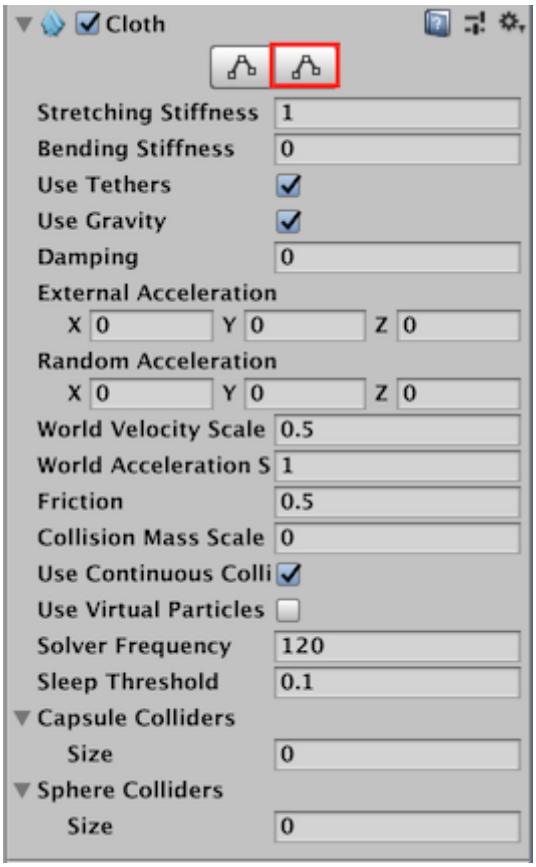
The Cloth Constraints Tool in Paint mode

Self collision and intercollision

Cloth collision makes character clothing and other fabrics in your game move more realistically. In Unity, a cloth has several cloth particles that handle collision. You can set up cloth particles for:

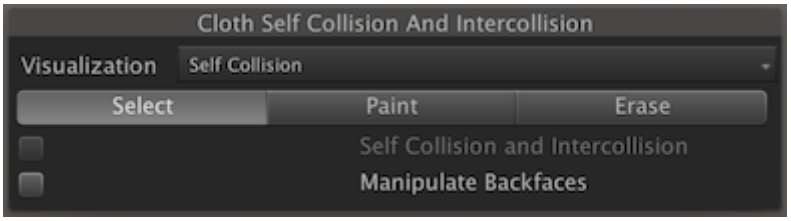
Self-collision, which prevents cloth from penetrating itself.
Intercollision, which allows cloth particles to collide with each other.

To set up the collision particles for a cloth, select the **Self Collision and Intercollision** button in the Cloth **inspector**:



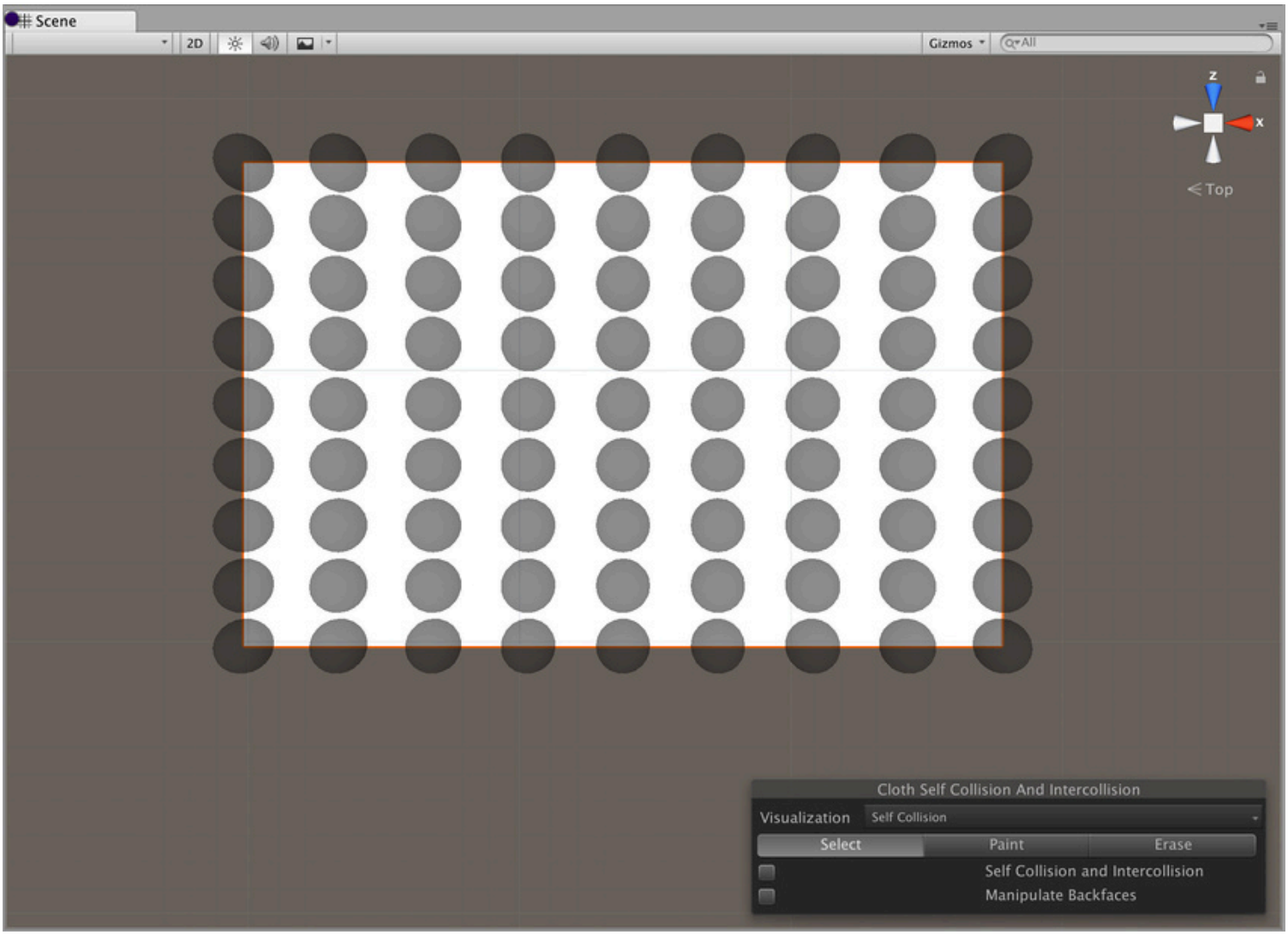
The Self Collision and Intercollision button in the Cloth Inspector

The **Cloth Self Collision And Intercollision** window appears in the Scene view:



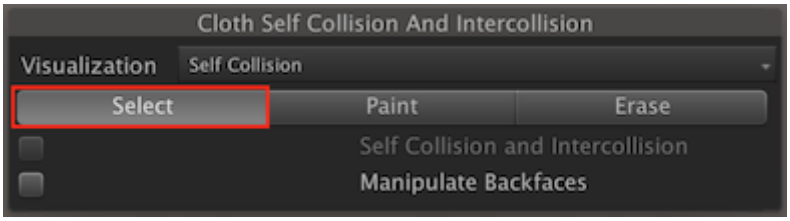
The Cloth Self Collision And Intercollision window

Cloth particles appear automatically for skinned Meshes with a Cloth component. Initially, none of the cloth particles are set to use collision. These unused particles appear black:



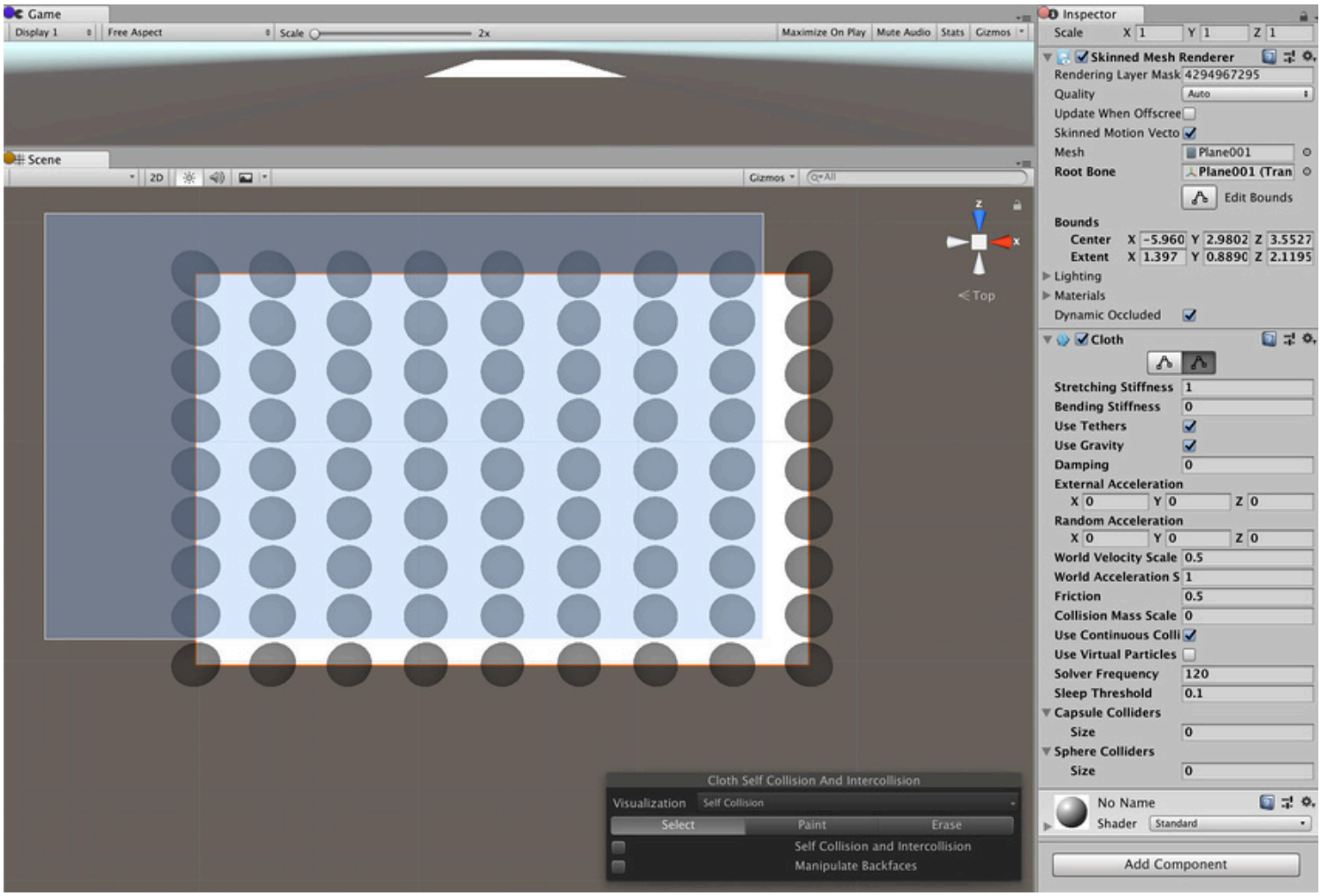
Unused cloth particles

To apply self-collision or intercollision, you need to select a single set of particles to apply collision to. To select a set of particles for collision, click the **Select** button:



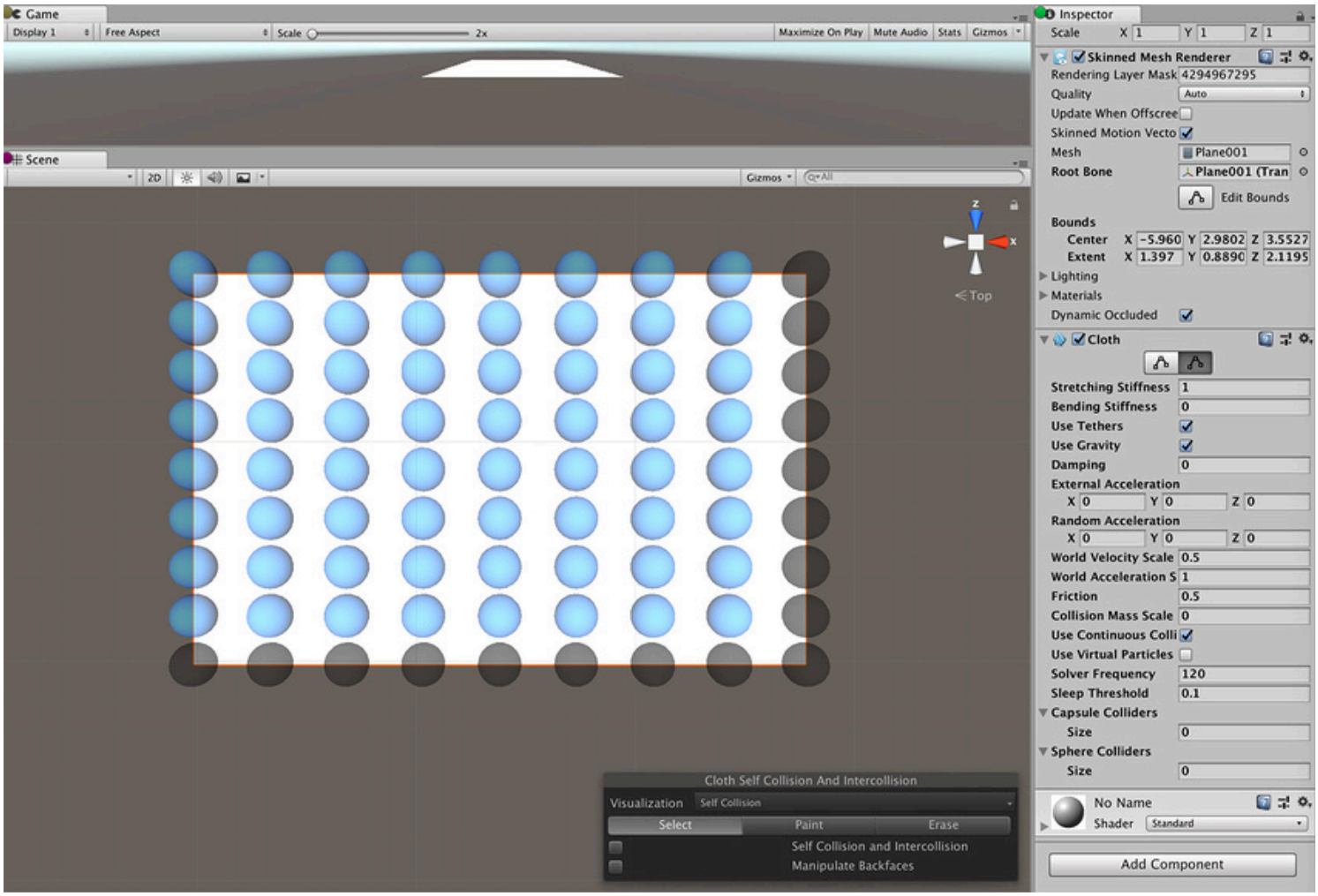
Select cloth particles button

Now left-click and drag to select the particles you want to apply collision to:



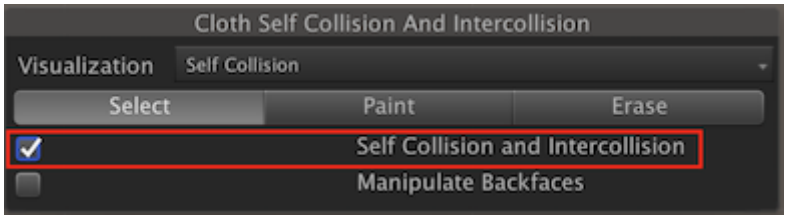
Selecting cloth particles using click and drag

The selected particles appear in blue:



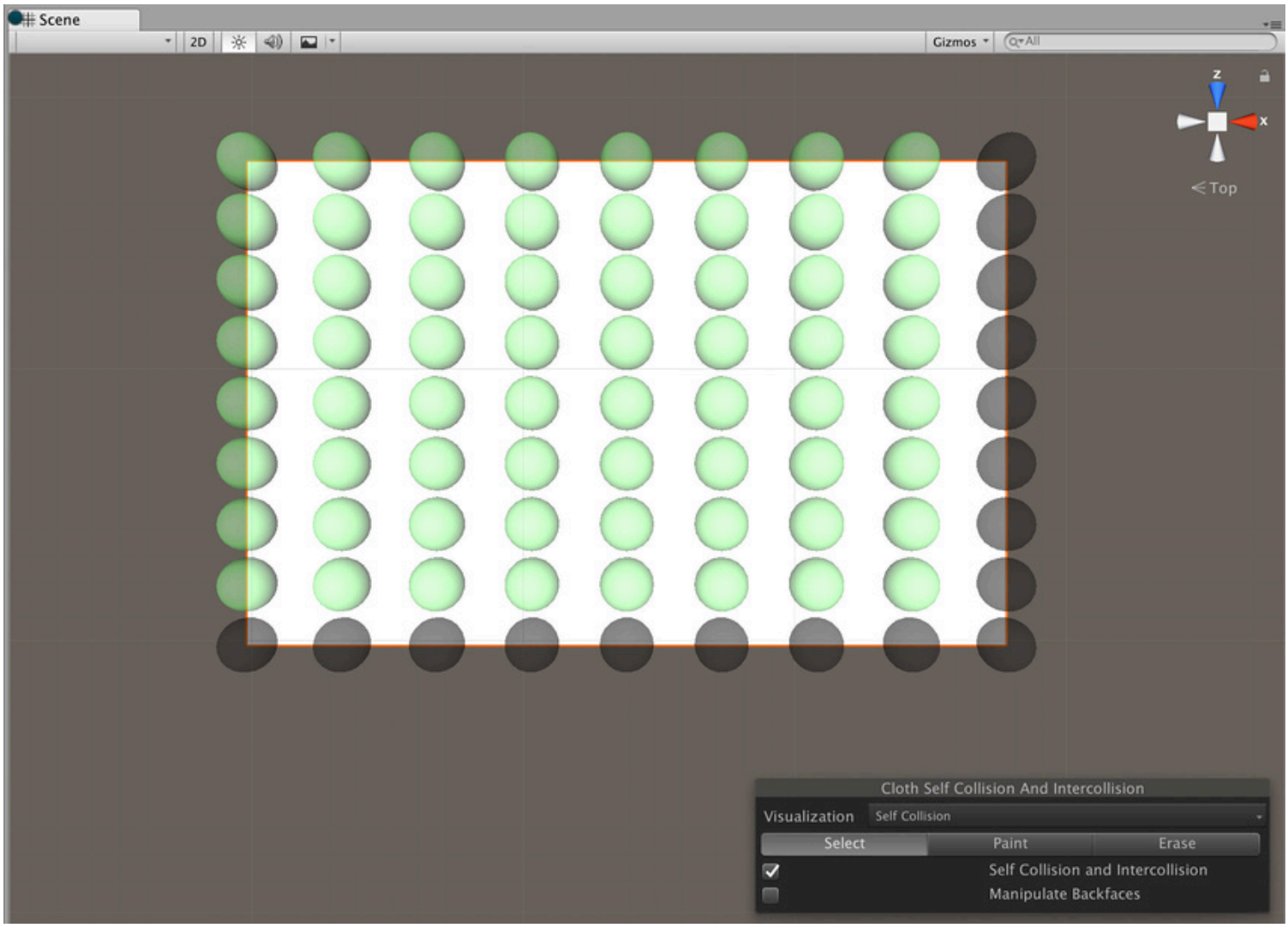
Selected cloth particles are blue

Tick the **Self Collision and Intercollision** checkbox to apply collision to the selected particles:



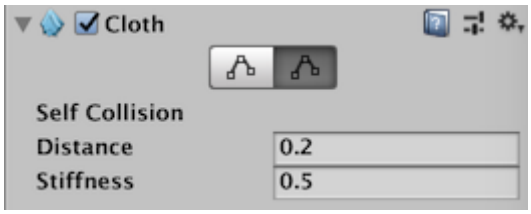
Self Collision and Intercollision tick box

The particles you specify for use in collision appear in green:



Selected particles are green

To enable the self collision behavior for a cloth, to go the **Self Collision** section of the Cloth Inspector window and set **Distance** and **Stiffness** to non-zero values:



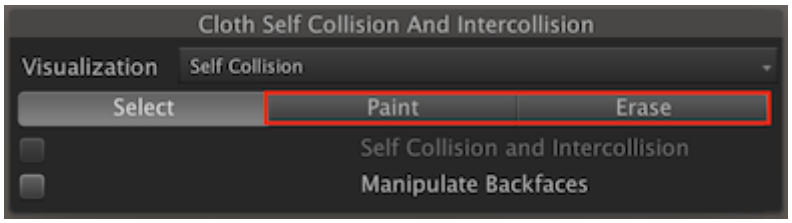
Self Collision parameters

Property:	*Function:*
Distance	The diameter of a sphere around each particle. Unity ensures that these spheres do not overlap during simulations. **Distance** should be smaller than the smallest distance between two particles in the configuration. If the distance is larger, self collision may violate some distance constraints and result in jittering.
Stiffness	How strong the separating impulse between particles should be. The cloth solver calculates this and it should be enough to keep the particles separated.

Self collision and intercollision can take a significant amount of the overall simulation time. Consider keeping the collision distance small and using self collision indices to reduce the number of particles that collide with each other.

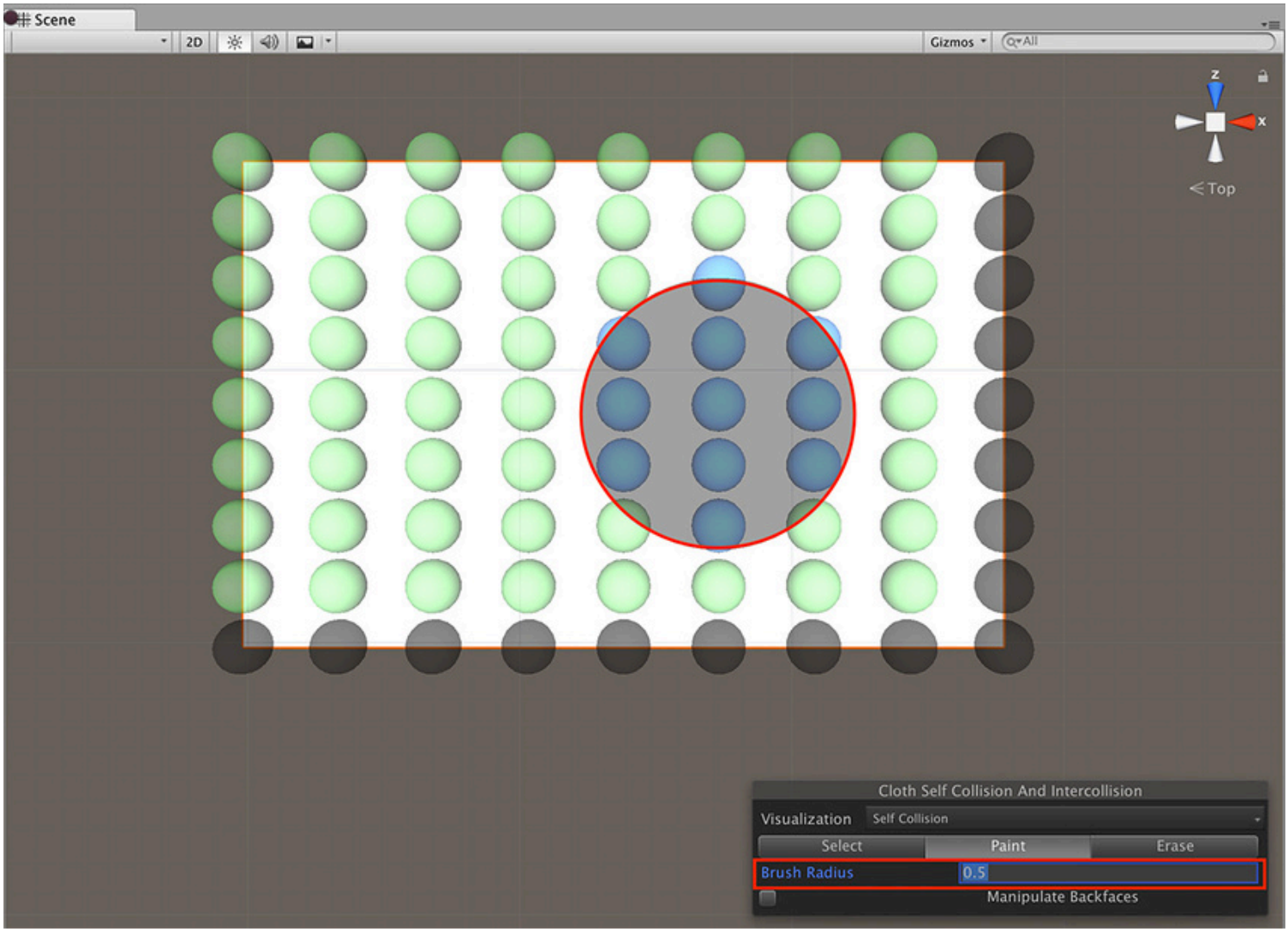
Self collision uses vertices, not triangles, so don't expect self collision to work perfectly for Meshes with triangles much larger than the cloth thickness.

Paint and **Erase** modes allow you to add or remove particles for use in collision by holding the left mouse button down and dragging individual cloth particles:



Self Collision parameters

When in **Paint** or **Erase** mode, particles specified for collision are green, unspecified particles are black, and particles underneath the brush are blue:

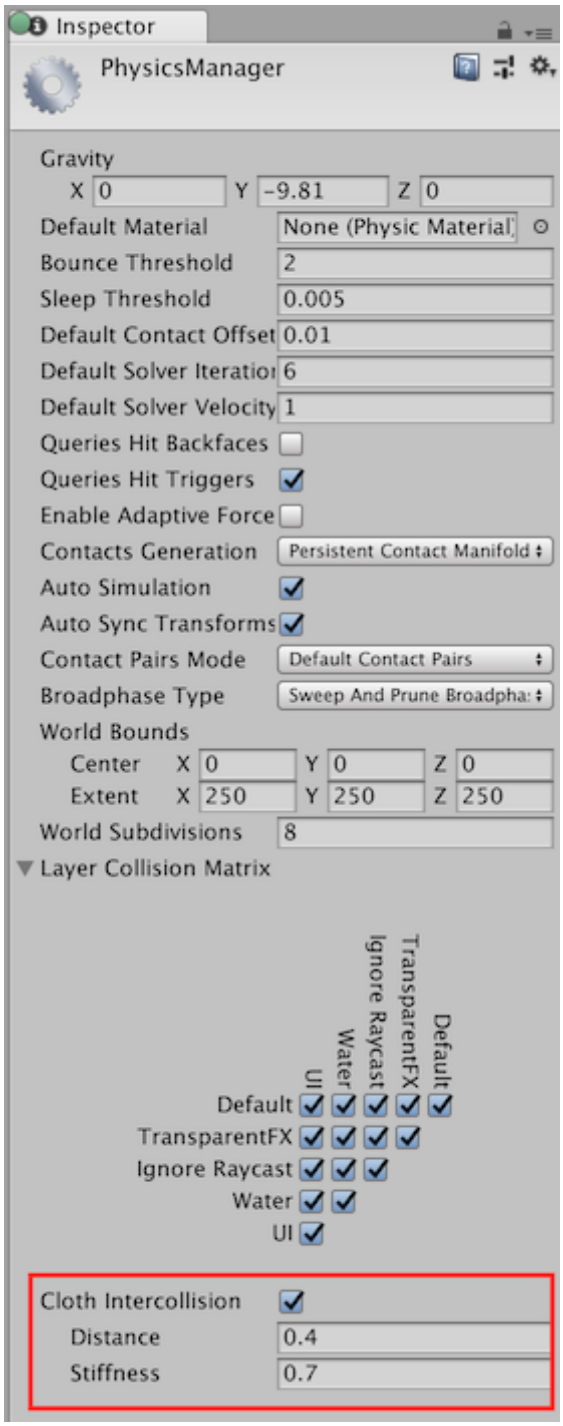


Particles being painted are blue

Cloth intercollision

You specify particles for intercollision in the same way as you specify particles for self collision, as described above. As with self collision, you specify one set of particles for intercollision.

To enable intercollision behavior, open the PhysicsManager inspector (**Edit** > **Project Settings** > **Physics**) and set **Distance** and **Stiffness** to non-zero values in the **Cloth InterCollision** section:



Particles being painted are blue

Cloth intercollision **Distance** and **Stiffness** properties have the same function as self collision Distance and Stiffness properties, which are described above.

Collider collision

Cloth is unable to simply collide with arbitrary world geometry, and now will only interact with the colliders specified in either the Capsule Colliders or Sphere Colliders arrays.

The sphere colliders array can contain either a single valid SphereCollider instance (with the second one being null), or a pair. In the former cases the ClothSphereColliderPair just represents a single sphere collider for the

cloth to collide against. In the latter case, it represents a conic capsule shape defined by the two spheres, and the cone connecting the two. Conic capsule shapes are useful for modelling limbs of a character.

2017–12–05 Page amended with limited editorial review

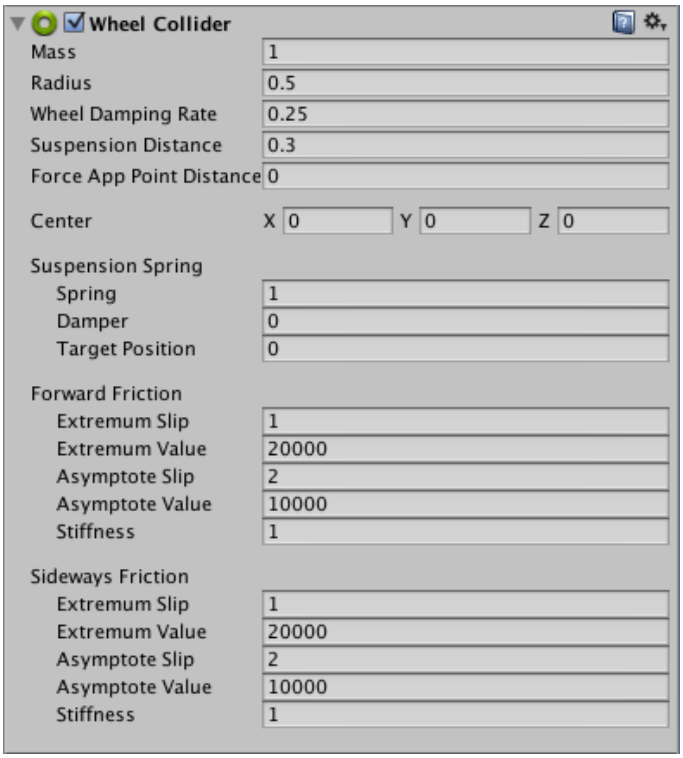
Cloth self collision and intercollision added in 2017.3

Wheel Collider

Leave feedback

SWITCH TO SCRIPTING

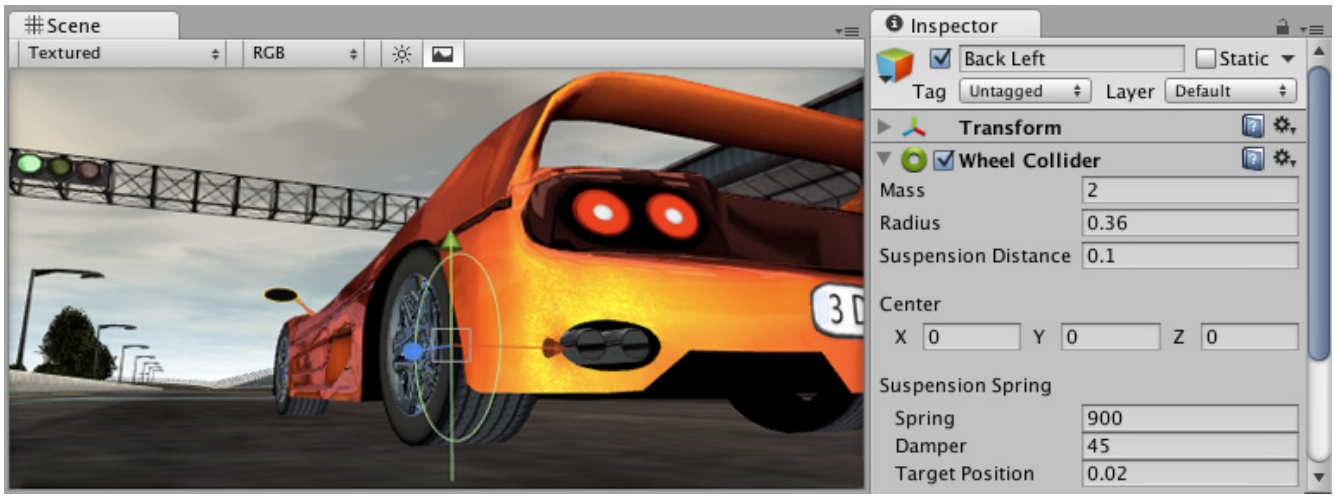
The **Wheel Collider** is a special **collider** for grounded vehicles. It has built-in **collision detection**, wheel physics, and a slip-based tire friction model. It can be used for objects other than wheels, but it is specifically designed for vehicles with wheels.



For guidance on using the Wheel Collider, see the Unity Wheel Collider tutorial.

Properties

Property:	*Function:*
Mass	The Mass of the wheel.
Radius	Radius of the wheel.
Wheel Damping Rate	This is a value of damping applied to a wheel.
Suspension Distance	Maximum extension distance of wheel suspension, measured in local space. Suspension always extends downwards through the local Y-axis.
Force App Point Distance	This parameter defines the point where the wheel forces will applied. This is expected to be in metres from the base of the wheel at rest position along the suspension travel direction. When `forceAppPointDistance = 0` the forces will be applied at the wheel base at rest. A better vehicle would have forces applied slightly below the vehicle centre of mass.
Center	Center of the wheel in object local space.
Suspension Spring	The suspension attempts to reach a **Target Position** by adding spring and damping forces.
Spring	Spring force attempts to reach the **Target Position**. A larger value makes the suspension reach the **Target Position** faster.
Damper	Dampens the suspension velocity. A larger value makes the **Suspension Spring** move slower.
Target Position	The suspension's rest distance along Suspension Distance. 1 maps to fully extended suspension, and 0 maps to fully compressed suspension. Default value is 0.5, which matches the behavior of a regular car's suspension.
Forward/Sideways Friction	Properties of tire friction when the wheel is rolling forward and sideways. See the *Wheel Friction Curves* section below.



The Wheel Collider Component. Car model courtesy of ATI Technologies Inc.

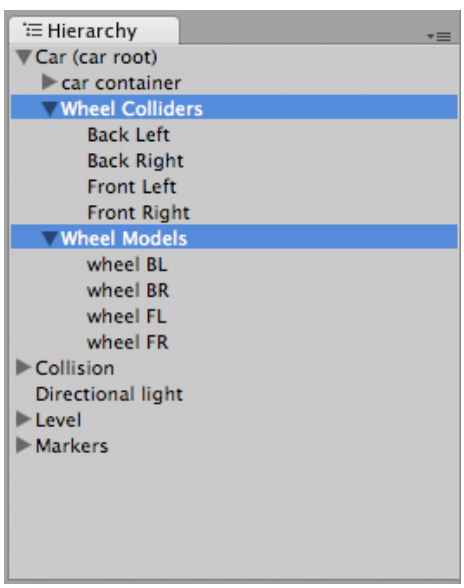
Details

The wheel's **collision** detection is performed by casting a ray from **Center** downwards through the local Y-axis. The wheel has a **Radius** and can extend downwards according to the **Suspension Distance**. The vehicle is controlled from scripting using different properties: **motorTorque**, **brakeTorque** and **steerAngle**. See the Wheel Collider scripting reference for more information.

The Wheel Collider computes friction separately from the rest of **physics engine**, using a slip-based friction model. This allows for more realistic behaviour but also causes Wheel Colliders to ignore standard Physic Material settings.

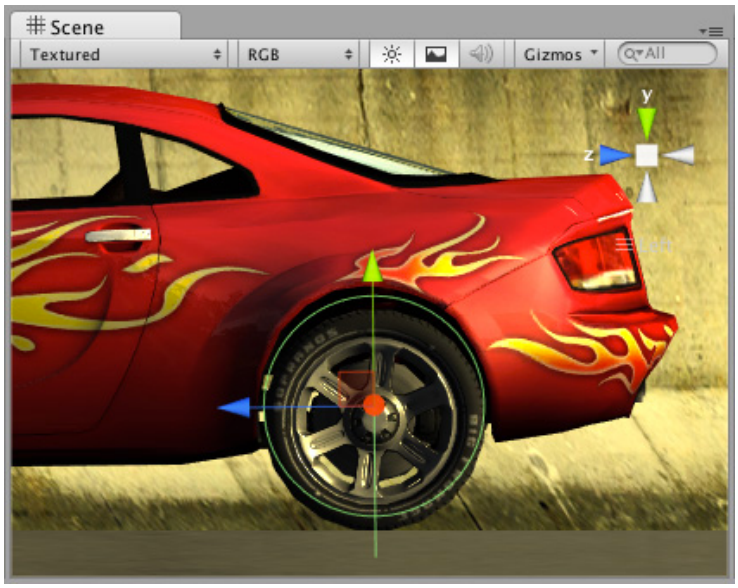
Wheel collider setup

You do not turn or roll WheelCollider objects to control the car - the objects that have the WheelCollider attached should always be fixed relative to the car itself. However, you might want to turn and roll the graphical wheel representations. The best way to do this is to setup separate objects for Wheel Colliders and visible wheels:



Wheel Colliders are separate from visible Wheel Models

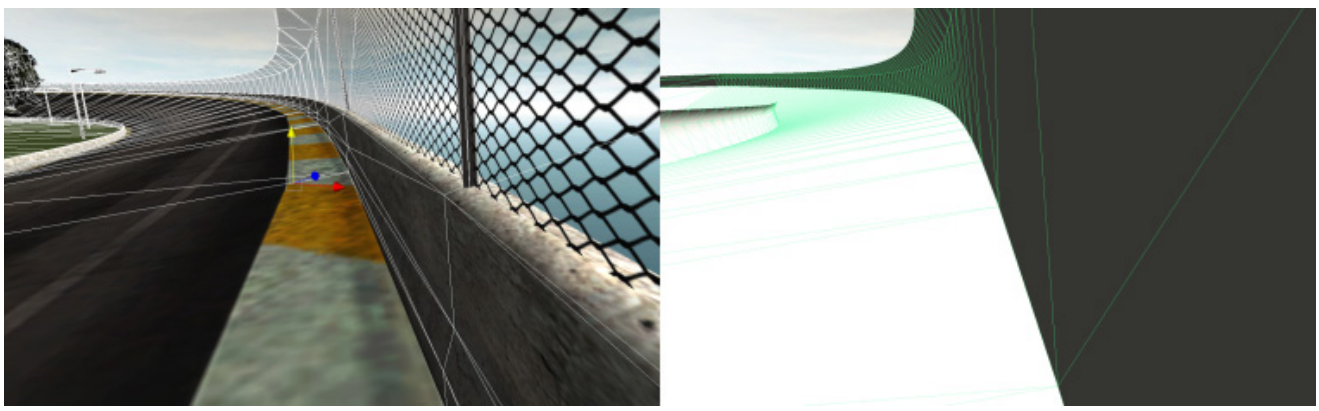
Note that the **gizmo** graphic for the WheelCollider's position is not updated in playmode:



Position of WheelCollider Gizmo in runtime using a suspension distance of 0.15

Collision geometry

Because cars can achieve large velocities, getting race track collision geometry right is very important. Specifically, the collision mesh should not have small bumps or dents that make up the visible models (e.g. fence poles). Usually a collision **mesh** for the race track is made separately from the visible **mesh**, making the collision **mesh** as smooth as possible. It also should not have thin objects - if you have a thin track border, make it wider in a collision **mesh** (or completely remove the other side if the car can never go there).

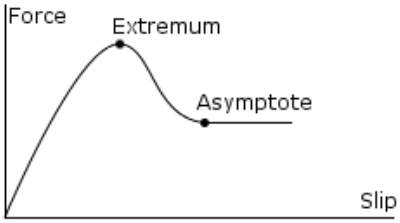


Visible geometry (left) is much more complex than collision geometry (right)

Wheel Friction Curves

Tire friction can be described by the *Wheel Friction Curve* shown below. There are separate curves for the wheel's forward (rolling) direction and sideways direction. In both directions it is first determined how much the tire is slipping (based on the speed difference between the tire's rubber and the road). Then this slip value is used to find out tire force exerted on the contact point.

The curve takes a measure of tire slip as an input and gives a force as output. The curve is approximated by a two-piece spline. The first section goes from *(0 , 0)* to *(**ExtremumSlip** , **ExtremumValue**)*, at which point the curve's tangent is zero. The second section goes from *(**ExtremumSlip** , **ExtremumValue**)* to *(**AsymptoteSlip** , **AsymptoteValue**)*, where curve's tangent is again zero:



Typical shape of a wheel friction curve

The property of real tires is that for low slip they can exert high forces, since the rubber compensates for the slip by stretching. Later when the slip gets really high, the forces are reduced as the tire starts to slide or spin. Thus, tire friction curves have a shape like in the image above.

Property:	*Function:*
Extremum Slip/Value	Curve's extremum point.
Asymptote Slip/Value	Curve's asymptote point.
Stiffness	Multiplier for the **Extremum Value** and **Asymptote Value** (default is 1). Changes the stiffness of the friction. Setting this to zero will completely disable all friction from the wheel. Usually you modify stiffness at runtime to simulate various ground materials from scripting.

Hints

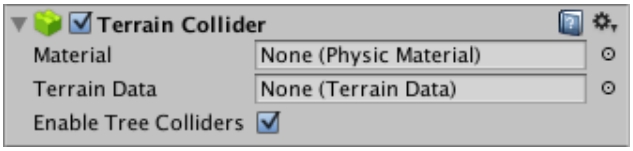
You might want to decrease physics timestep length in Time Manager to get more stable car physics, especially if it's a racing car that can achieve high velocities.
To keep a car from flipping over too easily you can lower its Rigidbody **center of mass** a bit from script, and apply "down pressure" force that depends on car velocity.

Terrain Collider

Leave feedback

SWITCH TO SCRIPTING

The **Terrain Collider** implements a **collision** surface with the same shape as the Terrain object it is attached to.



Properties

Property:	*Function:*
Material	Reference to the Physics Material that determines how this **Collider** interacts with others.
Terrain Data	The terrain data.
Enable Tree Colliders	When selected Tree Colliders will be enabled.

Details

You should note that versions of Unity before 5.0 had a **Smooth Sphere Collisions** property for the Terrain Collider in order to improve interactions between terrains and spheres. This property is now obsolete since the smooth interaction is standard behaviour for the **physics engine** and there is no particular advantage in switching it off.

Physic Material

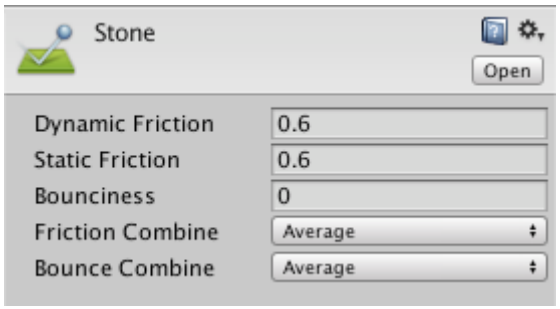
Leave feedback

SWITCH TO SCRIPTING

The **Physic Material** is used to adjust friction and bouncing effects of colliding objects.

To create a Physic Material select **Assets > Create > Physic Material** from the menu bar. Then drag the Physic Material from the Project View onto a **Collider** in the scene.

Properties



Property:	*Function:*
Dynamic Friction	The friction used when already moving. Usually a value from 0 to 1. A value of zero feels like ice, a value of 1 will make it come to rest very quickly unless a lot of force or gravity pushes the object.
Static Friction	The friction used when an object is laying still on a surface. Usually a value from 0 to 1. A value of zero feels like ice, a value of 1 will make it very hard to get the object moving.
Bounciness	How bouncy is the surface? A value of 0 will not bounce. A value of 1 will bounce without any loss of energy, certain approximations are to be expected though that might add small amounts of energy to the simulation.
Friction Combine	How the friction of two colliding objects is combined.
- **Average**	The two friction values are averaged.
- **Minimum**	The smallest of the two values is used.
- **Maximum**	The largest of the two values is used.
- **Multiply**	The friction values are multiplied with each other.
Bounce Combine	How the bounciness of two colliding objects is combined. It has the same modes as Friction Combine Mode

Details

Friction is the quantity which prevents surfaces from sliding off each other. This value is critical when trying to stack objects. Friction comes in two forms, dynamic and static. **Static friction** is used when the object is lying still. It will prevent the object from starting to move. If a large enough force is applied to the object it will start moving. At this point **Dynamic Friction** will come into play. **Dynamic Friction** will now attempt to slow down the object while in contact with another.

When two bodies are in contact, the same bounciness and friction effect is applied to both of them according to the chosen mode. There is a special case when the two colliders in contact have different combine modes set. In

this particular case, the function that has the highest priority is used. The priority order is as follows: **Average** < **Minimum** < **Multiply** < **Maximum**. For example, if one material has **Average** set but the other one has **Maximum**, then the combine function to be used is **Maximum**, since it has higher priority.

Please note that the friction model used by the Nvidia PhysX engine is tuned for performance and stability of simulation, and does not necessarily present a close approximation of real-world physics. In particular, contact surfaces which are larger than a single point (such as two boxes resting on each other) will be calculated as having two contact points, and will have friction forces twice as big as they would in real world physics. You may want to multiply your friction coefficients by 0.5 to get more realistic results in such a case.

The same logic applies to the bounciness model. Nvidia PhysX doesn't guarantee perfect energy conservation due to various simulation details such as position correction. So for example when the bounciness value of an object affected by gravity is 1 and is colliding with ground that has bounciness 1 expect the object to reach at higher positions than then initial one.

2017–07–17 Page amended with no editorial review

Updated functionality in 5.5

Physics HOWTOs

Leave feedback

This section contains a list of common physics-related tasks in Unity, and how to carry them out.

Ragdoll Wizard

Leave feedback

Unity has a simple wizard that lets you quickly create your own ragdoll. You simply have to drag the different limbs on the respective properties in the wizard. Then select create and Unity will automatically generate all **Colliders**, **Rigidbodies** and **Joints** that make up the Ragdoll for you.

Creating the Character

Ragdolls make use of **Skinned Meshes**, that is a character **mesh** rigged up with bones in the 3D modeling application. For this reason, you must build ragdoll characters in a 3D package like Autodesk® Maya® or Cinema4D.

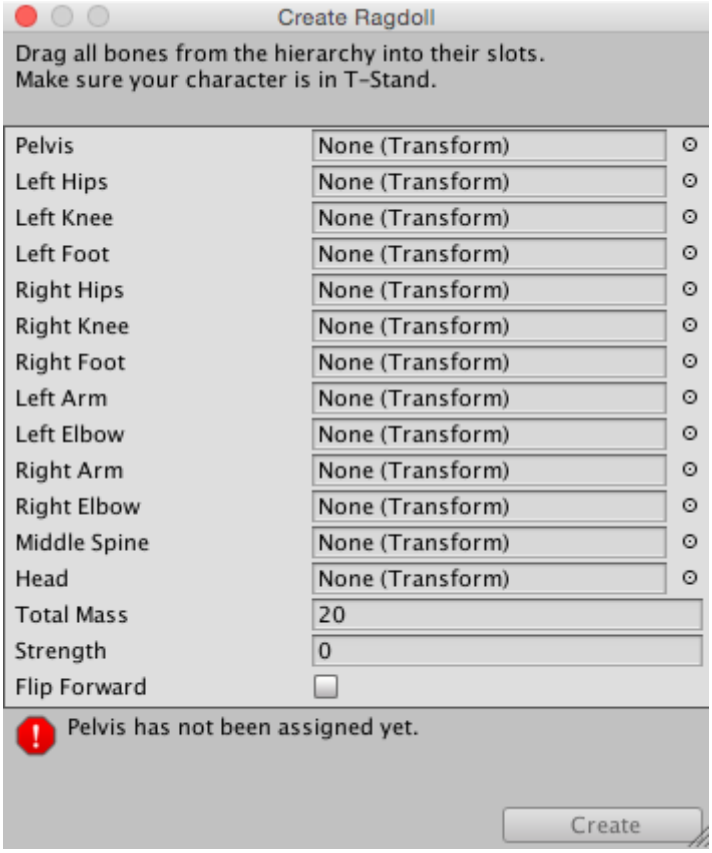
When you've created your character and rigged it, save the **asset** normally in your **Project Folder**. When you switch to Unity, you'll see the character **asset** file. Select that file and the **Import Settings** dialog will appear inside the **inspector**. Make sure that **Mesh Colliders** is not enabled.

Using the Wizard

It's not possible to make the actual source asset into a ragdoll. This would require modifying the source asset file, and is therefore impossible. You will make an instance of the character asset into a ragdoll, which can then be saved as a **Prefab** for re-use.

Create an instance of the character by dragging it from the **Project View** to the **Hierarchy View**. Expand its **Transform Hierarchy** by clicking the small arrow to the left of the instance's name in the Hierarchy. Now you are ready to start assigning your ragdoll parts.

Open the Ragdoll Wizard by choosing **GameObject > 3D Object > Ragdoll...** from the menu bar. You will now see the Wizard itself.



The Ragdoll Wizard

Assigning parts to the wizard should be self-explanatory. Drag the different Transforms of your character instance to the appropriate property on the wizard. This should be especially easy if you created the character asset yourself.

When you are done, click the **Create Button**. Now when you enter **Play Mode**, you will see your character go limp as a ragdoll.

The final step is to save the setup ragdoll as a Prefab. Choose **Assets -> Create -> Prefab** from the menu bar. You will see a New Prefab appear in the Project View. Rename it to “Ragdoll Prefab”. Drag the ragdoll character instance from the Hierarchy on top of the “Ragdoll Prefab”. You now have a completely set-up, re-usable ragdoll character to use as much as you like in your game.

Note

For **Character Joints** made with the Ragdoll wizard, take a note that the setup is made such that the joint’s Twist axis corresponds with the limb’s largest **swing axis**, the joint’s Swing 1 axis corresponds with limb’s smaller **swing axis** and joint’s Swing 2 is for twisting the limb. This naming scheme is for legacy reasons.

Joint and Ragdoll stability

Leave feedback

This page provides tips for improving Joint and Ragdoll stability.

Avoid small Joint angles of **Angular Y Limit** and **Angular Z Limit**. Depending on your setup, the minimum angles should be around 5 to 15 degrees in order to be stable. Instead of using a small angle, try setting the angle to zero. This locks the axis and provide a stable simulation.
Uncheck the Joint's **Enable Preprocessing** property. Disabling preprocessing can help prevent Joints from separating or moving erratically if they are forced into situations where there is no possible way to satisfy the Joint constraints. This can occur if Rigidbody **components** connected by Joints are pulled apart by static **collision** geometry (for example, spawning a Ragdoll partially inside a wall).
Under extreme circumstances (such as spawning partially inside a wall or pushed with a large force), the joint solver is unable to keep the Rigidbody components of a Ragdoll together. This can result in stretching. To handle this, enable projection on the Joints using either ConfigurableJoint.projectionMode or CharacterJoint.enableProjection.
If Rigidbody components connected with Joints are jittering, open the Physics Manager (**Edit** > **Project Settings** > **Physics**) and try increasing the **Default Solver Iterations** value to between 10 and 20.
If Rigidbody components connected with Joints are not accurately responding to bounces, open the Physics Manager (**Edit** > **Project Settings** > **Physics**) and try increasing the **Default Solver Velocity Iterations** value to between 10 and 20.
Never use direct Transform access with Kinematic Rigidbody components connected by Joints to other Rigidbody components. Doing so skips the step where PhysX computes internal velocities of corresponding Rigidbody components, making the solver provide unwanted results. A common example of bad practice is using direct Transform access in 2D projects to flip characters, by altering Transform.TransformDirection on the root boon of the rig. This behaves much better if you use Rigidbody2D.MovePosition and Rigidbody2D.MoveRotation instead.
Avoid large differences in the masses between Rigidbody components connected by Joints. It's okay to have one Rigidbody with twice as much mass as another, but when one mass is ten times larger than the other, the simulation can become jittery.
Try to avoid scaling different from 1 in the Transform containing Rigidbody or the Joint. The scaling might not be robust in all cases.
If Rigidbody components are overlapping when inserted into the world, and you cannot avoid the overlap, try lowering the Rigidbody.maxDepenetrationVelocity to make the Rigidbody components exit each other more smoothly.

Wheel Collider Tutorial

Leave feedback

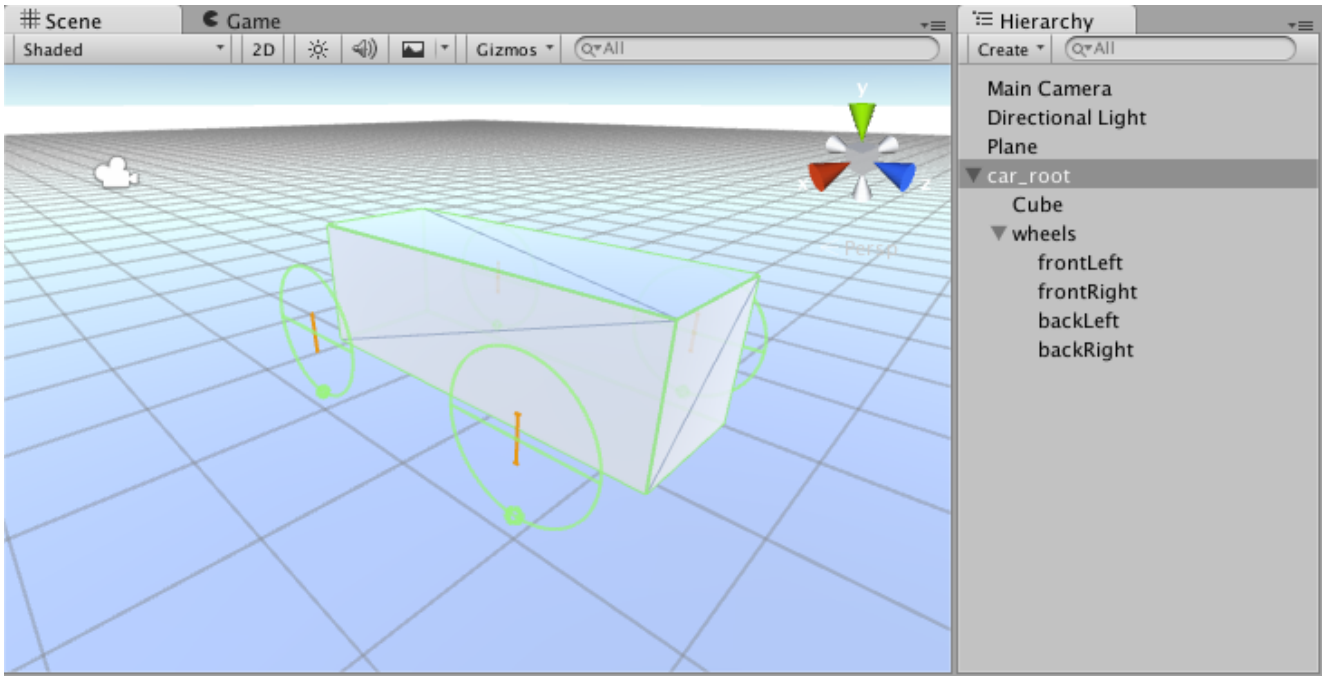
The Wheel **Collider component** is powered by the PhysX 3 Vehicles SDK.

This tutorial takes you through the process of creating a basic functioning car.

To start, select **GameObject** > **3D Object** > **Plane**. This is the ground the car is going to drive on. To keep it simple, make sure the ground has a Transform of **0** (on the Transform component in the Inspector Window, click the Settings cog and click **Reset**). Increase the Transform's Scale fields to **100** to make the Plane bigger.

Create a basic car skeleton

First, add a GameObject to act as the car root GameObject. To do this, go to **GameObject** > **Create Empty**. Change the GameObject's name to `car_root`.
Add a Physics 3D **Rigidbody** component to `car_root`. The default mass of 1kg is too light for the default suspension settings; change it to 1500kg to make it much heavier.
Next, create the car body Collider. Go to **GameObject** > **3D Object** > **Cube**. Make this cube a child GameObject under `car_root`. Reset the Transform to **0** to make it perfectly aligned in local space. The car is oriented along the Z axis, so set the Transform's **Z Scale** to **3**.
Add the wheels root. Select `car_root` and **GameObject** > **Create Empty Child**. Change the name to `wheels`. Reset the Transform on it. This GameObject is not mandatory, but it is useful for tuning and debugging later.
To create the first wheel, select the `wheels` GameObject, go to **GameObject** > **Create Empty Child**, and name it `frontLeft`. Reset the Transform, then set the Transform **Position X** to –1, **Y** to 0, and **Z** to 1. To add a Collider to the wheel, go to **Add component** > **Physics** > **Wheel Collider**.
Duplicate the `frontLeft` GameObject. Change the **Transform**'s **X** position to 1. Change the name to `frontRight`.
Select both the `frontLeft` and `frontRight` GameObjects. Duplicate them. Change the **Transform**'s **Z** position of both GameObjects to –1. Change the names to `rearLeft` and `rearRight` respectively.
Finally, select the `car_root` GameObject and use the Move Tool to raise it slightly above the ground.
Now you should be able to see something like this:

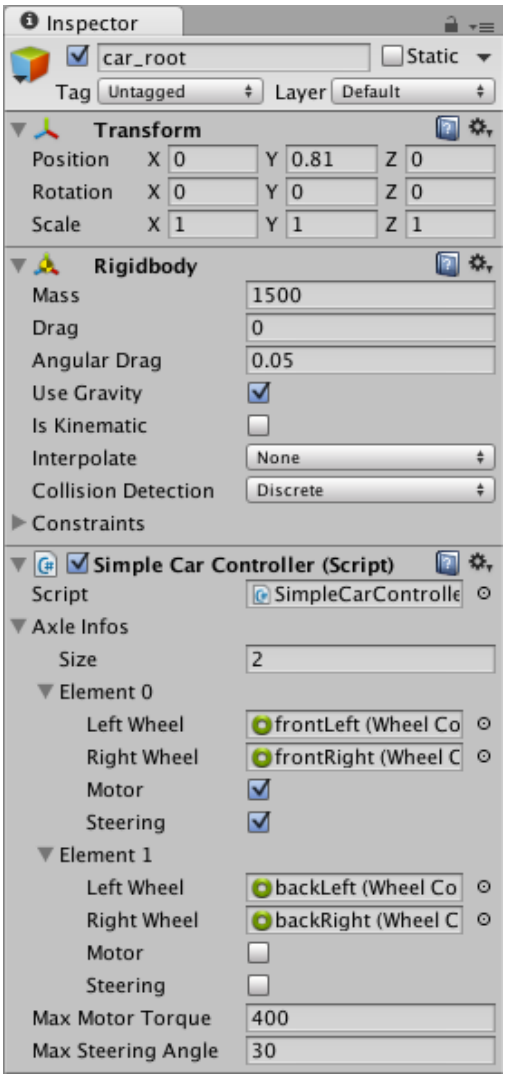


To make this car actually drivable, you need to write a controller for it. The following code sample works as a controller:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SimpleCarController : MonoBehaviour {
    public List<AxleInfo> axleInfos; // the information about each individual axle
    public float maxMotorTorque; // maximum torque the motor can apply to wheel
    public float maxSteeringAngle; // maximum steer angle the wheel can have

    public void FixedUpdate()
    {
        float motor = maxMotorTorque * Input.GetAxis("Vertical");
        float steering = maxSteeringAngle * Input.GetAxis("Horizontal");

        foreach (AxleInfo axleInfo in axleInfos) {
            if (axleInfo.steering) {
                axleInfo.leftWheel.steerAngle = steering;
                axleInfo.rightWheel.steerAngle = steering;
            }
            if (axleInfo.motor) {
                axleInfo.leftWheel.motorTorque = motor;
                axleInfo.rightWheel.motorTorque = motor;
            }
        }
    }
}

[System.Serializable]
public class AxleInfo {
    public WheelCollider leftWheel;
    public WheelCollider rightWheel;
    public bool motor; // is this wheel attached to motor?
    public bool steering; // does this wheel apply steer angle?
}
```

Create a new C# script (**Add Component** > **New Script**), on the `car_root` GameObject, copy this sample into the script file and save it. You can tune the script parameters as shown below; experiment with the settings and enter Play Mode to test the results.

The following settings are very effective as a car controller:



You can have up to 20 wheels on a single vehicle instance, with each of them applying steering, motor or braking torque.

Next, move on to visual wheels. As you can see, a Wheel Collider doesn't apply the simulated wheel position and rotation back to the Wheel Collider's Transform, so adding visual wheel requires some tricks.

You need some wheel geometry here. You can make a simple wheel shape out of a cylinder. There could be several approaches to adding visual wheels: making it so that you have to assign visual wheels manually in script properties, or writing some logic to find the corresponding visual wheel automatically. This tutorial follows the second approach. Attach the visual wheels to the Wheel Collider GameObjects.

Next, change the controller script:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

[System.Serializable]
public class AxleInfo {
    public WheelCollider leftWheel;
    public WheelCollider rightWheel;
    public bool motor;
    public bool steering;
}
```

```
public class SimpleCarController : MonoBehaviour {
    public List<AxleInfo> axleInfos;
    public float maxMotorTorque;
    public float maxSteeringAngle;

    // finds the corresponding visual wheel
    // correctly applies the transform
    public void ApplyLocalPositionToVisuals(WheelCollider collider)
    {
        if (collider.transform.childCount == 0) {
            return;
        }

        Transform visualWheel = collider.transform.GetChild(0);

        Vector3 position;
        Quaternion rotation;
        collider.GetWorldPose(out position, out rotation);

        visualWheel.transform.position = position;
        visualWheel.transform.rotation = rotation;
    }

    public void FixedUpdate()
    {
        float motor = maxMotorTorque * Input.GetAxis("Vertical");
        float steering = maxSteeringAngle * Input.GetAxis("Horizontal");

        foreach (AxleInfo axleInfo in axleInfos) {
            if (axleInfo.steering) {
                axleInfo.leftWheel.steerAngle = steering;
                axleInfo.rightWheel.steerAngle = steering;
            }
            if (axleInfo.motor) {
                axleInfo.leftWheel.motorTorque = motor;
                axleInfo.rightWheel.motorTorque = motor;
            }
            ApplyLocalPositionToVisuals(axleInfo.leftWheel);
            ApplyLocalPositionToVisuals(axleInfo.rightWheel);
        }
    }
}
```

One important parameter of the Wheel Collider component is **Force App Point Distance**. This is the distance from the base of the resting wheel to the point where the wheel forces are applied. The default value is 0, which means to apply the forces at the base of the resting wheel, but actually, it is wise to have this point located somewhere slightly below the car's centre of mass.

Note: To see the Wheel Collider in action, download the Vehicle Tools package, which includes tools to rig wheeled vehicles and create suspension for wheel colliders.

2017–11–24 Page amended with limited editorial review

Scripting

Leave feedback

Scripting is an essential ingredient in all games. Even the simplest game needs **scripts**, to respond to input from the player and arrange for events in the gameplay to happen when they should. Beyond that, **scripts** can be used to create graphical effects, control the physical behaviour of objects or even implement a custom AI system for characters in the game.

Scripting is a skill that takes some time and effort to learn. The intention of this section is not to teach you how to write script code from scratch, but rather to explain the main concepts that apply to scripting in Unity.

Related tutorials: Scripting

See the Knowledge Base Editor section for troubleshooting, tips and tricks.

Scripting Overview

Leave feedback

Although Unity uses an implementation of the standard Mono runtime for scripting, it still has its own practices and techniques for accessing the engine from **scripts**. This section explains how objects created in the Unity editor are controlled from **scripts**, and details the relationship between Unity's gameplay features and the Mono runtime.

Creating and Using Scripts

Leave feedback

The behavior of **GameObjects** is controlled by the **Components** that are attached to them. Although Unity's built-in Components can be very versatile, you will soon find you need to go beyond what they can provide to implement your own gameplay features. Unity allows you to create your own Components using **scripts**. These allow you to trigger game events, modify Component properties over time and respond to user input in any way you like.

Unity supports the C# programming language natively. C# (pronounced C-sharp) is an industry-standard language similar to Java or C++.

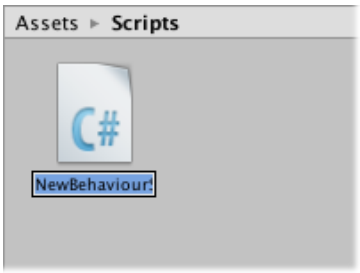
In addition to this, many other .NET languages can be used with Unity if they can compile a compatible DLL - see here for further details.

Learning the art of programming and the use of these particular languages is beyond the scope of this introduction. However, there are many books, tutorials and other resources for learning how to program with Unity. See the Learning section of our website for further details.

Creating Scripts

Unlike most other assets, scripts are usually created within Unity directly. You can create a new script from the Create menu at the top left of the Project panel or by selecting **Assets > Create > C# Script** from the main menu.

The new script will be created in whichever folder you have selected in the Project panel. The new script file's name will be selected, prompting you to enter a new name.



It is a good idea to enter the name of the new script at this point rather than editing it later. The name that you enter will be used to create the initial text inside the file, as described below.

Anatomy of a Script file

When you double-click a script Asset in Unity, it will be opened in a text editor. By default, Unity will use Visual Studio, but you can select any editor you like from the **External Tools** panel in Unity's preferences (go to **Unity** > **Preferences**).

The initial contents of the file will look something like this:

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
```

```
    }
}
```

A script makes its connection with the internal workings of Unity by implementing a class which derives from the built-in class called **MonoBehaviour**. You can think of a class as a kind of blueprint for creating a new Component type that can be attached to GameObjects. Each time you attach a script component to a GameObject, it creates a new instance of the object defined by the blueprint. The name of the class is taken from the name you supplied when the file was created. The class name and file name must be the same to enable the script component to be attached to a GameObject.

The main things to note, however, are the two functions defined inside the class. The **Update** function is the place to put code that will handle the frame update for the GameObject. This might include movement, triggering actions and responding to user input, basically anything that needs to be handled over time during gameplay. To enable the Update function to do its work, it is often useful to be able to set up variables, read preferences and make connections with other GameObjects before any game action takes place. The **Start** function will be called by Unity before gameplay begins (ie, before the Update function is called for the first time) and is an ideal place to do any initialization.

Note to experienced programmers: you may be surprised that initialization of an object is not done using a constructor function. This is because the construction of objects is handled by the editor and does not take place at the start of gameplay as you might expect. If you attempt to define a constructor for a script component, it will interfere with the normal operation of Unity and can cause major problems with the project.

Controlling a GameObject

As noted above, a script only defines a blueprint for a Component and so none of its code will be activated until an instance of the script is attached to a GameObject. You can attach a script by dragging the script **asset** to a GameObject in the hierarchy panel or to the **inspector** of the GameObject that is currently selected. There is also a Scripts submenu on the Component menu which will contain all the scripts available in the project, including those you have created yourself. The script instance looks much like any other Component in the **Inspector**:

Once attached, the script will start working when you press Play and run the game. You can check this by adding the following code in the Start function:-

```
// Use this for initialization
void Start ()
{
    Debug.Log("I am alive!");
}
```

Debug.Log is a simple command that just prints a message to Unity's console output. If you press Play now, you should see the message at the bottom of the main Unity editor window and in the Console window (menu: **Window** > **General** > **Console**).

2018–03–19 Page amended with limited editorial review

MonoDevelop replaced by Visual Studio from 2018.1

Variables and the Inspector

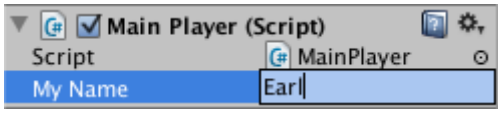
Leave feedback

When creating a script, you are essentially creating your own new type of **component** that can be attached to Game Objects just like any other **component**.

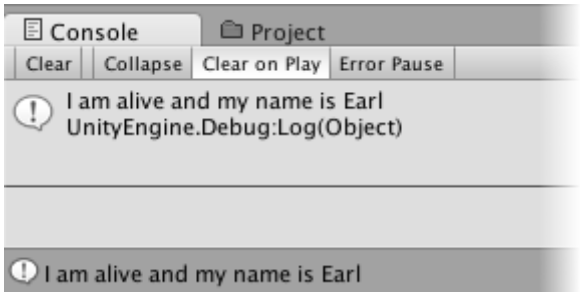
Just like other Components often have properties that are editable in the **inspector**, you can allow values in your script to be edited from the **Inspector** too.

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour
{
    public string myName;

    // Use this for initialization
    void Start ()
    {
        Debug.Log("I am alive and my name is " + myName);
    }
}
```

This code creates an editable field in the Inspector labelled “My Name”.



Unity creates the Inspector label by introducing a space wherever a capital letter occurs in the variable name. However, this is purely for display purposes and you should always use the variable name within your code. If you edit the name and then press Play, you will see that the message includes the text you entered.



In C#, you must declare a variable as public to see it in the Inspector.

Unity will actually let you change the value of a script’s variables while the game is running. This is very useful for seeing the effects of changes directly without having to stop and restart. When gameplay ends, the values of the variables will be reset to whatever they were before you pressed Play. This ensures that you are free to tweak your object’s settings without fear of doing any permanent damage.

Controlling GameObjects using components

Leave feedback

In the Unity Editor, you make changes to **Component** properties using the **Inspector**. So, for example, changes to the position values of the Transform **Component** will result in a change to the GameObject's position. Similarly, you can change the color of a Renderer's material or the mass of a **Rigidbody** with a corresponding effect on the appearance or behavior of the GameObject. For the most part, scripting is also about modifying **Component** properties to manipulate **GameObjects**. The difference, though, is that a script can vary a property's value gradually over time or in response to input from the user. By changing, creating and destroying objects at the right time, any kind of gameplay can be implemented.

Accessing components

The simplest and most common case is where a script needs access to other Components attached to the same GameObject. As mentioned in the Introduction section, a Component is actually an instance of a class so the first step is to get a reference to the Component instance you want to work with. This is done with the GetComponent function. Typically, you want to assign the Component object to a variable, which is done in C# using the following syntax:

```
void Start ()
{
    Rigidbody rb = GetComponent<Rigidbody>();
}
```

Once you have a reference to a Component instance, you can set the values of its properties much as you would in the Inspector:

```
void Start ()
{
    Rigidbody rb = GetComponent<Rigidbody>();

    // Change the mass of the object's Rigidbody.
    rb.mass = 10f;
}
```

An extra feature that is not available in the Inspector is the possibility of calling functions on Component instances:

```
void Start ()
{
    Rigidbody rb = GetComponent<Rigidbody>();

    // Add a force to the Rigidbody.
    rb.AddForce(Vector3.up * 10f);
}
```

Note also that there is no reason why you can't have more than one custom script attached to the same object. If you need to access one script from another, you can use GetComponent as usual and just use the name of the script class (or the filename) to specify the Component type you want.

If you attempt to retrieve a Component that hasn't actually been added to the GameObject then GetComponent will return null; you will get a null reference error at runtime if you try to change any values on a null object.

Accessing other objects

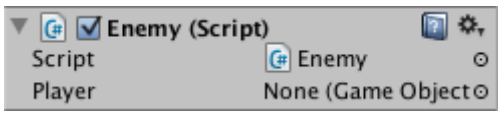
Although they sometimes operate in isolation, it is common for **scripts** to keep track of other objects. For example, a pursuing enemy might need to know the position of the player. Unity provides a number of different ways to retrieve other objects, each appropriate to certain situations.

Linking GameObjects with variables

The most straightforward way to find a related GameObject is to add a public GameObject variable to the script:

```
public class Enemy : MonoBehaviour
{
    public GameObject player;

    // Other variables and functions...
}
```

This variable will be visible in the Inspector like any other:



You can now drag an object from the **scene** or Hierarchy panel onto this variable to assign it. The GetComponent function and Component access variables are available for this object as with any other, so you can use code like the following:

```
public class Enemy : MonoBehaviour {
    public GameObject player;

    void Start() {
        // Start the enemy ten units behind the player character.
        transform.position = player.transform.position - Vector3.forward * 10f;
    }
}
```

Additionally, if declare a public variable of a Component type in your script, you can drag any GameObject that has that Component attached onto it. This will access the Component directly rather than the GameObject itself.

```
public Transform playerTransform;
```

Linking objects together with variables is most useful when you are dealing with individual objects that have permanent connections. You can use an array variable to link several objects of the same type, but the connections must still be made in the Unity editor rather than at runtime. It is often convenient to locate objects at runtime and Unity provides two basic ways to do this, as described below.

Finding child GameObjects

Sometimes, a game Scene makes use of a number of GameObjects of the same type, such as enemies, waypoints and obstacles. These may need to be tracked by a particular script that supervises or reacts to them (for example, all waypoints might need to be available to a pathfinding script). Using variables to link these GameObjects is a possibility but it makes the design process tedious if each new waypoint has to be dragged to a variable on a script. Likewise, if a waypoint is deleted, then it is a nuisance to have to remove the variable reference to the missing GameObject. In cases like this, it is often better to manage a set of GameObjects by making them all children of one parent GameObject. The child GameObjects can be retrieved using the parent's **Transform component** (because all GameObjects implicitly have a Transform):

```
using UnityEngine;

public class WaypointManager : MonoBehaviour {
    public Transform[] waypoints;

    void Start()
    {
        waypoints = new Transform[transform.childCount];
        int i = 0;
```

```
        foreach (Transform t in transform)
        {
            waypoints[i++] = t;
        }
    }
}
```

You can also locate a specific child object by name using the Transform.Find function:
`transform.Find("Gun");`

This can be useful when an object has a child that can be added and removed during gameplay. A weapon that can be picked up and put down is a good example of this.

Finding GameObjects by Name or Tag

It is always possible to locate GameObjects anywhere in the Scene hierarchy as long as you have some information to identify them. Individual objects can be retrieved by name using the GameObject.Find function:

```
GameObject player;

void Start()
{
    player = GameObject.Find("MainHeroCharacter");
}
```

An object or a collection of objects can also be located by their tag using the GameObject.FindWithTag and GameObject.FindGameObjectsWithTag functions:-

```
GameObject player;
GameObject[] enemies;

void Start()
{
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}
```

Event Functions

Leave feedback

A script in Unity is not like the traditional idea of a program where the code runs continuously in a loop until it completes its task. Instead, Unity passes control to a script intermittently by calling certain functions that are declared within it. Once a function has finished executing, control is passed back to Unity. These functions are known as event functions since they are activated by Unity in response to events that occur during gameplay. Unity uses a naming scheme to identify which function to call for a particular event. For example, you will already have seen the Update function (called before a frame update occurs) and the Start function (called just before the object's first frame update). Many more event functions are available in Unity; the full list can be found in the script reference page for the MonoBehaviour class along with details of their usage. The following are some of the most common and important events.

Regular Update Events

A game is rather like an animation where the animation frames are generated on the fly. A key concept in games programming is that of making changes to position, state and behavior of objects in the game just before each frame is rendered. The Update function is the main place for this kind of code in Unity. Update is called before the frame is rendered and also before animations are calculated.

```
void Update() {
    float distance = speed * Time.deltaTime * Input.GetAxis("Horizontal");
    transform.Translate(Vector3.right * distance);
}
```

The **physics engine** also updates in discrete time steps in a similar way to the frame **rendering**. A separate event function called FixedUpdate is called just before each physics update. Since the physics updates and frame updates do not occur with the same frequency, you will get more accurate results from physics code if you place it in the FixedUpdate function rather than Update.

```
void FixedUpdate() {
    Vector3 force = transform.forward * driveForce * Input.GetAxis("Vertical");
    rigidbody.AddForce(force);
}
```

It is also useful sometimes to be able to make additional changes at a point after the Update and FixedUpdate functions have been called for all objects in the **scene** and after all animations have been calculated. An example is where a **camera** should remain trained on a target object; the adjustment to the **camera**'s orientation must be made after the target object has moved. Another example is where the script code should override the effect of an

animation (say, to make the character's head look towards a target object in the scene). The LateUpdate function can be used for these kinds of situations.

```
void LateUpdate() {
    Camera.main.transform.LookAt(target.transform);
}
```

Initialization Events

It is often useful to be able to call initialization code in advance of any updates that occur during gameplay. The Start function is called before the first frame or physics update on an object. The Awake function is called for each object in the scene at the time when the scene loads. Note that although the various objects' Start and Awake functions are called in arbitrary order, all the Awakes will have finished before the first Start is called. This means that code in a Start function can make use of other initializations previously carried out in the Awake phase.

GUI events

Unity has a system for rendering GUI controls over the main action in the scene and responding to clicks on these controls. This code is handled somewhat differently from the normal frame update and so it should be placed in the OnGUI function, which will be called periodically.

```
void OnGUI() {
    GUI.Label(labelRect, "Game Over");
}
```

You can also detect mouse events that occur over a **GameObject** as it appears in the scene. This can be used for targeting weapons or displaying information about the character currently under the mouse pointer. A set of OnMouseXXX event functions (eg, OnMouseOver, OnMouseDown) is available to allow a script to react to user actions with the mouse. For example, if the mouse button is pressed while the pointer is over a particular object then an OnMouseDown function in that object's script will be called if it exists.

Physics events

The physics engine will report **collisions** against an object by calling event functions on that object's script. The OnCollisionEnter, OnCollisionStay and OnCollisionExit functions will be called as contact is made, held and broken. The corresponding OnTriggerEnter, OnTriggerStay and OnTriggerExit functions will be called when the object's **collider** is configured as a Trigger (ie, a collider that simply detects when something enters it rather than reacting physically). These functions may be called several times in succession if more than one contact is detected during the physics update and so a parameter is passed to the function giving details of the collision (position, identity of the incoming object, etc).

```
void OnCollisionEnter(otherObj: Collision) {
    if (otherObj.tag == "Arrow") {
        ApplyDamage(10);
    }
}
```

Time and Framerate Management

Leave feedback

The Update function allows you to monitor inputs and other events regularly from a script and take appropriate action. For example, you might move a character when the “forward” key is pressed. An important thing to remember when handling time-based actions like this is that the game’s framerate is not constant and neither is the length of time between Update function calls.

As an example of this, consider the task of moving an object forward gradually, one frame at a time. It might seem at first that you could just shift the object by a fixed distance each frame:

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public float distancePerFrame;

    void Update() {
        transform.Translate(0, 0, distancePerFrame);
    }
}


//JS script example
var distancePerFrame: float;

function Update() {
    transform.Translate(0, 0, distancePerFrame);
}
```

However, given that the frame time is not constant, the object will appear to move at an irregular speed. If the frame time is 10 milliseconds then the object will step forward by *distancePerFrame* one hundred times per second. But if the frame time increases to 25 milliseconds (due to CPU load, say) then it will only step forward forty times a second and therefore cover less distance. The solution is to scale the size of the movement by the frame time which you can read from the Time.deltaTime property:

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public float distancePerSecond;
```

```
    void Update() {
        transform.Translate(0, 0, distancePerSecond * Time.deltaTime);
    }
}


//JS script example
var distancePerSecond: float;

function Update() {
    transform.Translate(0, 0, distancePerSecond * Time.deltaTime);
}
```

Note that the movement is now given as *distancePerSecond* rather than *distancePerFrame*. As the framerate changes, the size of the movement step will change accordingly and so the object's speed will be constant.

Fixed Timestep

Unlike the main frame update, Unity's physics system *does* work to a **fixed timestep**, which is important for the accuracy and consistency of the simulation. At the start of the physics update, Unity sets an "alarm" by adding the **fixed timestep** value onto the time when the last physics update ended. The physics system will then perform calculations until the alarm goes off.

You can change the size of the fixed timestep from the Time Manager and you can read it from a script using the Time.fixedDeltaTime property. Note that a lower value for the timestep will result in more frequent physics updates and more precise simulation but at the cost of greater CPU load. You probably won't need to change the default fixed timestep unless you are placing high demands on the **physics engine**.

Maximum Allowed Timestep

The fixed timestep keeps the physical simulation accurate in real time but it can cause problems in cases where the game makes heavy use of physics and the gameplay framerate has also become low (due to a large number of objects in play, say). The main frame update processing has to be "squeezed" in between the regular physics updates and if there is a lot of processing to do then several physics updates can take place during a single frame. Since the frame time, positions of objects and other properties are frozen at the start of the frame, the graphics can get out of sync with the more frequently updated physics.

Naturally, there is only so much CPU power available but Unity has an option to let you effectively slow down physics time to let the frame processing catch up. The *Maximum Allowed Timestep* setting (in the Time Manager) puts a limit on the amount of time Unity will spend processing physics and FixedUpdate calls during a given frame update. If a frame update takes longer than *Maximum Allowed Timestep* to process, the physics engine will "stop time" and let the frame processing catch up. Once the frame update has finished, the physics will resume as though no time has passed since it was stopped. The result of this is that **rigidbodies** will not move perfectly in real time as they usually do but will be slowed slightly. However, the physics "clock" will still track them as though

they were moving normally. The slowing of physics time is usually not noticeable and is an acceptable trade-off against gameplay performance.

Time Scale

For special effects, such as “bullet-time”, it is sometimes useful to slow the passage of game time so that animations and script responses happen at a reduced rate. Furthermore, you may sometimes want to freeze game time completely, as when the game is paused. Unity has a *Time Scale* property that controls how fast game time proceeds relative to real time. If the scale is set to 1.0 then game time matches real time. A value of 2.0 makes time pass twice as quickly in Unity (ie, the action will be speeded-up) while a value of 0.5 will slow gameplay down to half speed. A value of zero will make time “stop” completely. Note that the time scale doesn’t actually slow execution but simply changes the time step reported to the Update and FixedUpdate functions via Time.deltaTime and Time.fixedDeltaTime. The Update function is likely to be called more often than usual when game time is slowed down but the *deltaTime* step reported each frame will simply be reduced. Other script functions are not affected by the time scale so you can, for example, display a GUI with normal interaction when the game is paused.

The Time Manager has a property to let you set the time scale globally but it is generally more useful to set the value from a script using the Time.timeScale property:

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void Pause() {
        Time.timeScale = 0;
    }

    void Resume() {
        Time.timeScale = 1;
    }
}

//JS script example
function Pause() {
    Time.timeScale = 0;
}

function Resume() {
    Time.timeScale = 1;
}
```

Capture Framerate

A very special case of time management is where you want to record gameplay as a video. Since the task of saving screen images takes considerable time, the usual framerate of the game will be drastically reduced if you attempt to do this during normal gameplay. This will result in a video that doesn't reflect the true performance of the game.

Fortunately, Unity provides a Capture Framerate property that lets you get around this problem. When the property's value is set to anything other than zero, game time will be slowed and the frame updates will be issued at precise regular intervals. The interval between frames is equal to 1 / Time.captureFramerate, so if the value is set to 5.0 then updates occur every fifth of a second. With the demands on framerate effectively reduced, you have time in the Update function to save screenshots or take other actions:

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    // Capture frames as a screenshot sequence. Images are
    // stored as PNG files in a folder - these can be combined into
    // a movie using image utility software (eg, QuickTime Pro).
    // The folder to contain our screenshots.
    // If the folder exists we will append numbers to create an empty folder.
    string folder = "ScreenshotFolder";
    int frameRate = 25;

    void Start () {
        // Set the playback framerate (real time will not relate to game time af
        Time.captureFramerate = frameRate;

        // Create the folder
        System.IO.Directory.CreateDirectory(folder);
    }

    void Update () {
        // Append filename to folder name (format is '0005 shot.png"')
        string name = string.Format("{0}/{1:D04} shot.png", folder, Time.frameCo

        // Capture the screenshot to the specified file.
        Application.CaptureScreenshot(name);
    }
}

//JS script example

// Capture frames as a screenshot sequence. Images are
// stored as PNG files in a folder - these can be combined into
// a movie using image utility software (eg, QuickTime Pro).
```

```
// The folder to contain our screenshots.
// If the folder exists we will append numbers to create an empty folder.
var folder = "ScreenshotFolder";
var frameRate = 25;


function Start () {
    // Set the playback framerate (real time will not relate to game time after
    Time.captureFramerate = frameRate;

    // Create the folder
    System.IO.Directory.CreateDirectory(folder);
}

function Update () {
    // Append filename to folder name (format is '0005 shot.png"')
    var name = String.Format("{0}/{1:D04} shot.png", folder, Time.frameCount );

    // Capture the screenshot to the specified file.
    Application.CaptureScreenshot(name);
}
```

Although the video recorded using this technique typically looks very good, the game can be hard to play when slowed-down drastically. You may need to experiment with the value of Time.captureFramerate to allow ample recording time without unduly complicating the task of the test player.

Creating and Destroying GameObjects

Leave feedback

Some games keep a constant number of objects in the **scene**, but it is very common for characters, treasures and other object to be created and removed during gameplay. In Unity, a **GameObject** can be created using the Instantiate function which makes a new copy of an existing object:

```
public GameObject enemy;

void Start() {
    for (int i = 0; i < 5; i++) {
        Instantiate(enemy);
    }
}
```

Note that the object from which the copy is made doesn't have to be present in the scene. It is more common to use a **prefab** dragged to a public variable from the Project panel in the editor. Also, instantiating a GameObject will copy all the **Components** present on the original.

There is also a Destroy function that will destroy an object after the frame update has finished or optionally after a short time delay:

```
void OnCollisionEnter(Collision otherObj) {
    if (otherObj.gameObject.tag == "Missile") {
        Destroy(gameObject,.5f);
    }
}
```

Note that the Destroy function can destroy individual components without affecting the GameObject itself. A common mistake is to write something like:

```
 Destroy(this);
```

...which will actually just destroy the script component that calls it rather than destroying the GameObject the script is attached to.

Coroutines

Leave feedback

When you call a function, it runs to completion before returning. This effectively means that any action taking place in a function must happen within a single frame update; a function call can't be used to contain a procedural animation or a sequence of events over time. As an example, consider the task of gradually reducing an object's alpha (opacity) value until it becomes completely invisible.

```
void Fade()
{
    for (float f = 1f; f >= 0; f -= 0.1f)
    {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
    }
}
```

As it stands, the Fade function will not have the effect you might expect. In order for the fading to be visible, the alpha must be reduced over a sequence of frames to show the intermediate values being rendered. However, the function will execute in its entirety within a single frame update. The intermediate values will never be seen and the object will disappear instantly.

It is possible to handle situations like this by adding code to the Update function that executes the fade on a frame-by-frame basis. However, it is often more convenient to use a coroutine for this kind of task.

A coroutine is like a function that has the ability to pause execution and return control to Unity but then to continue where it left off on the following frame. In C#, a coroutine is declared like this:

```
IEnumerator Fade()
{
    for (float f = 1f; f >= 0; f -= 0.1f)
    {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield return null;
    }
}
```

It is essentially a function declared with a return type of IEnumerator and with the yield return statement included somewhere in the body. The yield return line is the point at which execution will pause and be resumed the following frame. To set a coroutine running, you need to use the StartCoroutine function:

```
void Update()
{
    if (Input.GetKeyDown("f"))
    {
        StartCoroutine("Fade");
    }
}
```

You will notice that the loop counter in the Fade function maintains its correct value over the lifetime of the coroutine. In fact any variable or parameter will be correctly preserved between yields.

By default, a coroutine is resumed on the frame after it yields but it is also possible to introduce a time delay using WaitForSeconds:

```
IEnumerator Fade()
{
    for (float f = 1f; f >= 0; f -= 0.1f)
    {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield return new WaitForSeconds(.1f);
    }
}
```

This can be used as a way to spread an effect over a period of time, but it is also a useful optimization. Many tasks in a game need to be carried out periodically and the most obvious way to do this is to include them in the Update function. However, this function will typically be called many times per second. When a task doesn't need to be repeated quite so frequently, you can put it in a coroutine to get an update regularly but not every single frame. An example of this might be an alarm that warns the player if an enemy is nearby. The code might look something like this:

```
function ProximityCheck()
{
    for (int i = 0; i < enemies.Length; i++)
```

```
    {
        if (Vector3.Distance(transform.position, enemies[i].transform.position)
                return true;
        }
    }

    return false;
}
```

If there are a lot of enemies then calling this function every frame might introduce a significant overhead. However, you could use a coroutine to call it every tenth of a second:

```
IEnumerator DoCheck()
{
    for(;;)
    {
        ProximityCheck;
        yield return new WaitForSeconds(.1f);
    }
}
```

This would greatly reduce the number of checks carried out without any noticeable effect on gameplay.

Note: Coroutines are not stopped when a MonoBehaviour is disabled, but only when it is definitely destroyed. You can stop a Coroutine using MonoBehaviour.StopCoroutine and MonoBehaviour.StopAllCoroutines. Coroutines are also stopped when the MonoBehaviour is destroyed.

Namespaces

Leave feedback

As projects become larger and the number of **scripts** increases, the likelihood of having clashes between script class names grows ever greater. This is especially true when several programmers are working on different aspects of the game separately and will eventually combine their efforts in one project. For example, one programmer may be writing the code to control the main player character while another writes the equivalent code for the enemy. Both programmers may choose to call their main script class *Controller*, but this will cause a clash when their projects are combined.

To some extent, this problem can be avoided by adopting a naming convention or by renaming classes whenever a clash is discovered (eg, the classes above could be given names like *PlayerController* and *EnemyController*). However, this is troublesome when there are several classes with clashing names or when variables are declared using those names - each mention of the old class name must be replaced for the code to compile.

The C# language offers a feature called **namespaces** that solves this problem in a robust way. A namespace is simply a collection of classes that are referred to using a chosen prefix on the class name. In the example below, the classes *Controller1* and *Controller2* are members of a namespace called *Enemy*:

```
namespace Enemy {
    public class Controller1 : MonoBehaviour {
        ...
    }

    public class Controller2 : MonoBehaviour {
        ...
    }
}
```

In code, these classes are referred to as `Enemy.Controller1` and `Enemy.Controller2`, respectively. This is better than renaming the classes insofar as the namespace declaration can be bracketed around existing class declarations (ie, it is not necessary to change the names of all the classes individually). Furthermore, you can use multiple bracketed namespace sections around classes wherever they occur, even if those classes are in different source files.

You can avoid having to type the namespace prefix repeatedly by adding a **using** directive at the top of the file.

```
using Enemy;
```

This line indicates that where the class names *Controller1* and *Controller2* are found, they should be taken to mean *Enemy.Controller1* and *Enemy.Controller2*, respectively. If the script also needs to refer to classes with the same

name from a different namespace (one called *Player*, say), then the prefix can still be used. If two namespaces that contain clashing class names are imported with using directives at the same time, the compiler will report an error.

Attributes

Leave feedback

Attributes are markers that can be placed above a class, property or function in a script to indicate special behaviour. For example, you can add the `HideInInspector` attribute above a property declaration to prevent the **Inspector** from showing the property, even if it is public. C# contains attribute names within square brackets, like so:

```
[HideInInspector]
public float strength;
```

Unity provides a number of attributes which are listed in the API Reference documentation:

For UnityEngine attributes, see AddComponentMenu and sibling pages
For UnityEditor attributes, see CallbackOrderAttribute and sibling pages
There are also attributes defined in the .NET libraries which might sometimes be useful in Unity code. See Microsoft's documentation on Attributes for more information.

Note: Do not use the ThreadStatic attribute defined in the .NET library; it causes a crash if you add it to a Unity script.

Execution Order of Event Functions

Leave feedback

In Unity scripting, there are a number of event functions that get executed in a predetermined order as a script executes. This execution order is described below:

First Scene Load

These functions get called when a **scene** starts (once for each object in the scene).

Awake: This function is always called before any Start functions and also just after a **prefab** is instantiated. (If a GameObject is inactive during start up Awake is not called until it is made active.)
OnEnable: (only called if the Object is active): This function is called just after the object is enabled. This happens when a MonoBehaviour instance is created, such as when a level is loaded or a **GameObject** with the script **component** is instantiated.
OnLevelWasLoaded: This function is executed to inform the game that a new level has been loaded.
Note that for objects added to the scene, the Awake and OnEnable functions for *all* **scripts** will be called before Start, Update, **etc** are called for any of them. Naturally, this cannot be enforced when an object is instantiated during gameplay.

Editor

Reset: Reset is called to initialize the script's properties when it is first attached to the object and also when the *Reset* command is used.

Before the first frame update

Start: Start is called before the first frame update only if the script instance is enabled.
For objects added to the scene, the Start function will be called on all scripts before Update, etc are called for any of them. Naturally, this cannot be enforced when an object is instantiated during gameplay.

In between frames

OnApplicationPause: This is called at the end of the frame where the pause is detected, effectively between the normal frame updates. One extra frame will be issued after **OnApplicationPause** is called to allow the game to show graphics that indicate the paused state.

Update Order

When you're keeping track of game logic and interactions, animations, **camera** positions, etc., there are a few different events you can use. The common pattern is to perform most tasks inside the **Update** function, but there are also other functions you can use.

FixedUpdate: FixedUpdate is often called more frequently than **Update**. It can be called multiple times per frame, if the frame rate is low and it may not be called between frames at all if the frame rate is high. All physics calculations and updates occur immediately after **FixedUpdate**. When applying movement calculations inside **FixedUpdate**, you do not need to multiply your values by **Time.deltaTime**. This is because **FixedUpdate** is called on a reliable timer, independent of the frame rate.

Update: Update is called once per frame. It is the main workhorse function for frame updates.

LateUpdate: LateUpdate is called once per frame, after **Update** has finished. Any calculations that are performed in **Update** will have completed when **LateUpdate** begins. A common use for **LateUpdate** would be a following third-person camera. If you make your character move and turn inside **Update**, you can perform all camera movement and rotation calculations in **LateUpdate**. This will ensure that the character has moved completely before the camera tracks its position.

Rendering

OnPreCull: Called before the camera culls the scene. Culling determines which objects are visible to the camera. OnPreCull is called just before culling takes place.
OnBecameVisible/OnBecameInvisible: Called when an object becomes visible/invisible to any camera.
OnWillRenderObject: Called **once** for each camera if the object is visible.
OnPreRender: Called before the camera starts **rendering** the scene.
OnRenderObject: Called after all regular scene rendering is done. You can use GL class or Graphics.DrawMeshNow to draw custom geometry at this point.
OnPostRender: Called after a camera finishes rendering the scene.
OnRenderImage: Called after scene rendering is complete to allow post-processing of the image, see Post-processing Effects.
OnGUI: Called multiple times per frame in response to GUI events. The Layout and Repaint events are processed first, followed by a Layout and keyboard/mouse event for each input event.
OnDrawGizmos Used for drawing **Gizmos** in the **scene view** for visualisation purposes.

Coroutines

Normal coroutine updates are run after the Update function returns. A coroutine is a function that can suspend its execution (yield) until the given YieldInstruction finishes. Different uses of Coroutines:

yield The coroutine will continue after all Update functions have been called on the next frame.
yield WaitForSeconds Continue after a specified time delay, after all Update functions have been called for the frame
yield WaitForFixedUpdate Continue after all FixedUpdate has been called on all scripts
yield WWW Continue after a WWW download has completed.
yield StartCoroutine Chains the coroutine, and will wait for the MyFunc coroutine to complete first.

When the Object is Destroyed

OnDestroy: This function is called after all frame updates for the last frame of the object's existence (the object might be destroyed in response to Object.Destroy or at the closure of a scene).

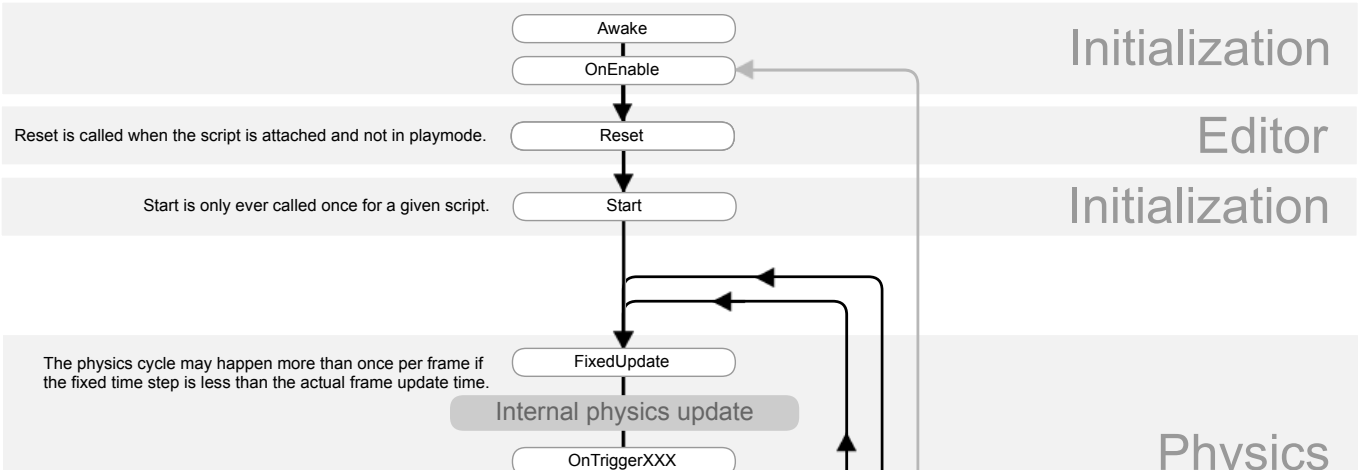
When Quitting

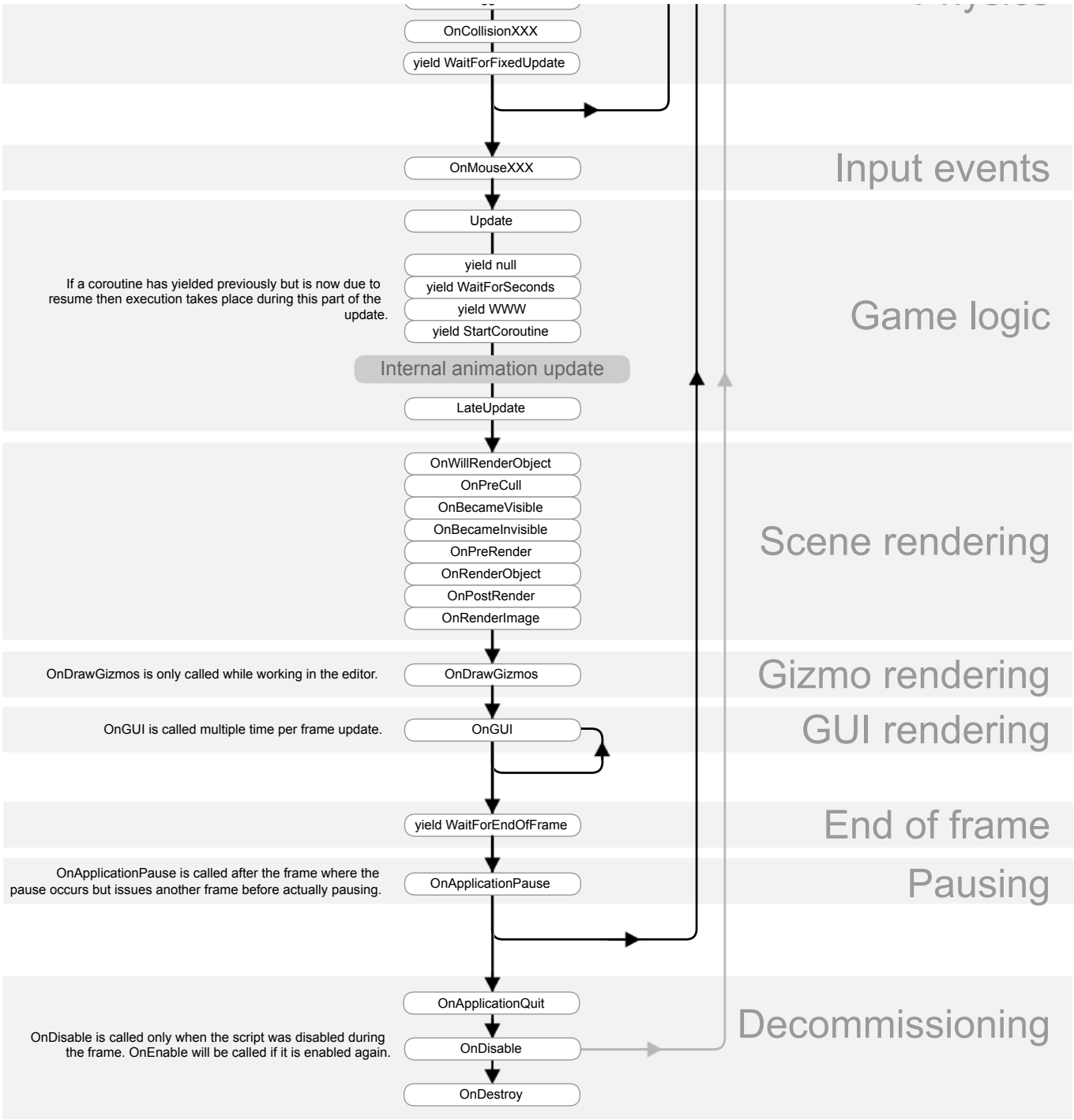
These functions get called on all the active objects in your scene:

OnApplicationQuit: This function is called on all game objects before the application is quit. In the editor it is called when the user stops playmode.
OnDisable: This function is called when the behaviour becomes disabled or inactive.

Script Lifecycle Flowchart

The following diagram summarises the ordering and repetition of event functions during a script's lifetime.





Note: Some browsers do not support SVG image files. If the image above does not display properly (for example, if you cannot see any text), please try another browser, such as Google Chrome or Mozilla Firefox.

Understanding Automatic Memory Management

Leave feedback

When an object, string or array is created, the memory required to store it is allocated from a central pool called the **heap**. When the item is no longer in use, the memory it once occupied can be reclaimed and used for something else. In the past, it was typically up to the programmer to allocate and release these blocks of heap memory explicitly with the appropriate function calls. Nowadays, runtime systems like Unity's Mono engine manage memory for you automatically. Automatic memory management requires less coding effort than explicit allocation/release and greatly reduces the potential for memory leakage (the situation where memory is allocated but never subsequently released).

Value and Reference Types

When a function is called, the values of its parameters are copied to an area of memory reserved for that specific call. Data types that occupy only a few bytes can be copied very quickly and easily. However, it is common for objects, strings and arrays to be much larger and it would be very inefficient if these types of data were copied on a regular basis. Fortunately, this is not necessary; the actual storage space for a large item is allocated from the heap and a small "pointer" value is used to remember its location. From then on, only the pointer need be copied during parameter passing. As long as the runtime system can locate the item identified by the pointer, a single copy of the data can be used as often as necessary.

Types that are stored directly and copied during parameter passing are called value types. These include integers, floats, booleans and Unity's struct types (eg, **Color** and **Vector3**). Types that are allocated on the heap and then accessed via a pointer are called reference types, since the value stored in the variable merely "refers" to the real data. Examples of reference types include objects, strings and arrays.

Allocation and Garbage Collection

The memory manager keeps track of areas in the heap that it knows to be unused. When a new block of memory is requested (say when an object is instantiated), the manager chooses an unused area from which to allocate the block and then removes the allocated memory from the known unused space. Subsequent requests are handled the same way until there is no free area large enough to allocate the required block size. It is highly unlikely at this point that all the memory allocated from the heap is still in use. A reference item on the heap can only be accessed as long as there are still reference variables that can locate it. If all references to a memory block are gone (ie, the reference variables have been reassigned or they are local variables that are now out of scope) then the memory it occupies can safely be reallocated.

To determine which heap blocks are no longer in use, the memory manager searches through all currently active reference variables and marks the blocks they refer to as "live". At the end of the search, any space between the live blocks is considered empty by the memory manager and can be used for subsequent allocations. For obvious reasons, the process of locating and freeing up unused memory is known as garbage collection (or GC for short).

Optimization

Garbage collection is automatic and invisible to the programmer but the collection process actually requires significant CPU time behind the **scenes**. When used correctly, automatic memory management will generally equal or beat manual allocation for overall performance. However, it is important for the programmer to avoid mistakes that will trigger the collector more often than necessary and introduce pauses in execution.

There are some infamous algorithms that can be GC nightmares even though they seem innocent at first sight. Repeated string concatenation is a classic example:

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void ConcatExample(int[] intArray) {
        string line = intArray[0].ToString();

        for (i = 1; i < intArray.Length; i++) {
            line += ", " + intArray[i].ToString();
        }

        return line;
    }
}


//JS script example
function ConcatExample(intArray: int[]) {
    var line = intArray[0].ToString();

    for (i = 1; i < intArray.Length; i++) {
        line += ", " + intArray[i].ToString();
    }

    return line;
}
```

The key detail here is that the new pieces don't get added to the string in place, one by one. What actually happens is that each time around the loop, the previous contents of the line variable become dead - a whole new string is allocated to contain the original piece plus the new part at the end. Since the string gets longer with increasing values of i, the amount of heap space being consumed also increases and so it is easy to use up hundreds of bytes of free heap space each time this function is called. If you need to concatenate many strings together then a much better option is the Mono library's System.Text.StringBuilder class.

However, even repeated concatenation won't cause too much trouble unless it is called frequently, and in Unity that usually implies the frame update. Something like:

```
//C# script example
using UnityEngine;
```

```
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public GUIText scoreBoard;
    public int score;

    void Update() {
        string scoreText = "Score: " + score.ToString();
        scoreBoard.text = scoreText;
    }
}


//JS script example
var scoreBoard: GUIText;
var score: int;

function Update() {
    var scoreText: String = "Score: " + score.ToString();
    scoreBoard.text = scoreText;
}
```

...will allocate new strings each time Update is called and generate a constant trickle of new garbage. Most of that can be saved by updating the text only when the score changes:

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public GUIText scoreBoard;
    public string scoreText;
    public int score;
    public int oldScore;

    void Update() {
        if (score != oldScore) {
            scoreText = "Score: " + score.ToString();
            scoreBoard.text = scoreText;
            oldScore = score;
        }
    }
}
```

```
//JS script example
var scoreBoard: GUIText;
var scoreText: String;
var score: int;
var oldScore: int;

function Update() {
    if (score != oldScore) {
        scoreText = "Score: " + score.ToString();
        scoreBoard.text = scoreText;
        oldScore = score;
    }
}
```

Another potential problem occurs when a function returns an array value:

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    float[] RandomList(int numElements) {
        var result = new float[numElements];

        for (int i = 0; i < numElements; i++) {
            result[i] = Random.value;
        }

        return result;
    }
}


//JS script example
function RandomList(numElements: int) {
    var result = new float[numElements];

    for (i = 0; i < numElements; i++) {
        result[i] = Random.value;
    }
```

```
    return result;
}
```

This type of function is very elegant and convenient when creating a new array filled with values. However, if it is called repeatedly then fresh memory will be allocated each time. Since arrays can be very large, the free heap space could get used up rapidly, resulting in frequent garbage collections. One way to avoid this problem is to make use of the fact that an array is a reference type. An array passed into a function as a parameter can be modified within that function and the results will remain after the function returns. A function like the one above can often be replaced with something like:

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void RandomList(float[] arrayToFill) {
        for (int i = 0; i < arrayToFill.Length; i++) {
            arrayToFill[i] = Random.value;
        }
    }
}


//JS script example
function RandomList(arrayToFill: float[]) {
    for (i = 0; i < arrayToFill.Length; i++) {
        arrayToFill[i] = Random.value;
    }
}
```

This simply replaces the existing contents of the array with new values. Although this requires the initial allocation of the array to be done in the calling code (which looks slightly inelegant), the function will not generate any new garbage when it is called.

Disabling garbage collection

If you are using the Mono or **IL2CPP scripting backend**, you can avoid CPU spikes during garbage collection by disabling garbage collection at run time. When you disable garbage collection, memory usage never decreases because the garbage collector does not collect objects that no longer have any references. In fact, memory usage can only ever increase when you disable garbage collection. To avoid increased memory usage over time, take

care when managing memory. Ideally, allocate all memory before you disable the garbage collector and avoid additional allocations while it is disabled.

For more details on how to enable and disable garbage collection at run time, see the GarbageCollector Scripting API page.

Requesting a Collection

As mentioned above, it is best to avoid allocations as far as possible. However, given that they can't be completely eliminated, there are two main strategies you can use to minimise their intrusion into gameplay:-

Small heap with fast and frequent garbage collection

This strategy is often best for games that have long periods of gameplay where a smooth framerate is the main concern. A game like this will typically allocate small blocks frequently but these blocks will be in use only briefly. The typical heap size when using this strategy on **iOS** is about 200KB and garbage collection will take about 5ms on an iPhone 3G. If the heap increases to 1MB, the collection will take about 7ms. It can therefore be advantageous sometimes to request a garbage collection at a regular frame interval. This will generally make collections happen more often than strictly necessary but they will be processed quickly and with minimal effect on gameplay:

```
if (Time.frameCount % 30 == 0)
{
   System.GC.Collect();
}
```

However, you should use this technique with caution and check the **profiler** statistics to make sure that it is really reducing collection time for your game.

Large heap with slow but infrequent garbage collection

This strategy works best for games where allocations (and therefore collections) are relatively infrequent and can be handled during pauses in gameplay. It is useful for the heap to be as large as possible without being so large as to get your app killed by the OS due to low system memory. However, the Mono runtime avoids expanding the heap automatically if at all possible. You can expand the heap manually by preallocating some placeholder space during startup (ie, you instantiate a "useless" object that is allocated purely for its effect on the memory manager):

```
//C# script example
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    void Start() {
        var tmp = new System.Object[1024];
```

```
        // make allocations in smaller blocks to avoid them to be treated in a s
        for (int i = 0; i < 1024; i++)
            tmp[i] = new byte[1024];

        // release reference
        tmp = null;
    }
}


//JS script example
function Start() {
    var tmp = new System.Object[1024];

    // make allocations in smaller blocks to avoid them to be treated in a speci
        for (var i : int = 0; i < 1024; i++)
        tmp[i] = new byte[1024];

    // release reference
        tmp = null;
}
```

A sufficiently large heap should not get completely filled up between those pauses in gameplay that would accommodate a collection. When such a pause occurs, you can request a collection explicitly:

```
System.GC.Collect();
```

Again, you should take care when using this strategy and pay attention to the profiler statistics rather than just assuming it is having the desired effect.

Reusable Object Pools

There are many cases where you can avoid generating garbage simply by reducing the number of objects that get created and destroyed. There are certain types of objects in games, such as projectiles, which may be encountered over and over again even though only a small number will ever be in play at once. In cases like this, it is often possible to reuse objects rather than destroy old ones and replace them with new ones.

Further Information

Memory management is a subtle and complex subject to which a great deal of academic effort has been devoted. If you are interested in learning more about it then memorymanagement.org is an excellent resource, listing many publications and online articles. Further information about object pooling can be found on the Wikipedia page and also at Sourcemaking.com.

2018–09–19 Page amended with limited editorial review

Ability to disable garbage collection on Mono and IL2CPP scripting backends added in Unity 2018.3

Platform dependent compilation

Leave feedback

Unity includes a feature called **Platform Dependent Compilation**. This consists of some preprocessor directives that let you partition your **scripts** to compile and execute a section of code exclusively for one of the supported platforms.

You can run this code within the Unity Editor, so you can compile the code specifically for your target platform and test it in the Editor!

Platform #define directives

The platform #define directives that Unity supports for your scripts are as follows:

Property:	Function:
UNITY_EDITOR	#define directive for calling Unity Editor scripts from your game code.
UNITY_EDITOR_WIN	#define directive for Editor code on Windows.
UNITY_EDITOR_OSX	#define directive for Editor code on Mac OS X.
UNITY_STANDALONE_OSX	#define directive for compiling/executing code specifically for Mac OS X (including Universal, PPC and Intel architectures).
UNITY_STANDALONE_WIN	#define directive for compiling/executing code specifically for Windows standalone applications.
UNITY_STANDALONE_LINUX	#define directive for compiling/executing code specifically for Linux standalone applications.
UNITY_STANDALONE	#define directive for compiling/executing code for any standalone platform (Mac OS X, Windows or Linux).
UNITY_WII	#define directive for compiling/executing code for the Wii console.
UNITY_IOS	#define directive for compiling/executing code for the iOS platform.
UNITY_IPHONE	Deprecated. Use **UNITY_IOS** instead.
UNITY_ANDROID	#define directive for the Android platform.
UNITY_PS4	#define directive for running PlayStation 4 code.
UNITY_XBOXONE	#define directive for executing **Xbox One** code.
UNITY_TIZEN	#define directive for the Tizen platform.
UNITY_TVOS	#define directive for the Apple TV platform.
UNITY_WSA	#define directive for **Universal Windows Platform**. Additionally, **NETFX_CORE** is defined when compiling C# files against .NET Core and using .NET **scripting backend**.
UNITY_WSA_10_0	#define directive for Universal Windows Platform. Additionally **WINDOWS_UWP** is defined when compiling C# files against .NET Core.
UNITY_WINRT	Same as **UNITY_WSA**.
UNITY_WINRT_10_0	Equivalent to **UNITY_WSA_10_0**
UNITY_WEBGL	#define directive for **WebGL**.
UNITY_FACEBOOK	#define directive for the Facebook platform (WebGL or Windows standalone).

Property:	Function:
UNITY_ADS	#define directive for calling Unity Ads methods from your game code. Version 5.2 and above.
UNITY_ANALYTICS	#define directive for calling Unity **Analytics** methods from your game code. Version 5.2 and above.
UNITY_ASSERTIONS	#define directive for assertions control process.

From Unity 2.6.0 onwards, you can compile code selectively. The options available depend on the version of the Editor that you are working on. Given a version number **X.Y.Z** (for example, 2.6.0), Unity exposes three global #define directives in the following formats: **UNITY_X**, **UNITY_X_Y** and **UNITY_X_Y_Z**.

Here is an example of #define directives exposed in Unity 5.0.1:

UNITY_5	#define directive for the release version of Unity 5, exposed in every 5.X.Y release.
UNITY_5_0	#define directive for the major version of Unity 5.0, exposed in every 5.0.Z release.
UNITY_5_0_1	#define directive for the minor version of Unity 5.0.1.

Starting from Unity 5.3.4, you can compile code selectively based on the earliest version of Unity required to compile or execute a given portion of code. Given the same version format as above (**X.Y.Z**), Unity exposes one global #define in the format **UNITY_X_Y_OR_NEWER**, that can be used for this purpose.

The supported #define directives are:

ENABLE_MONO	Scripting backend #define for Mono.
ENABLE_IL2CPP	Scripting backend #define for **IL2CPP**.
ENABLE_DOTNET	Scripting backend #define for .NET.
NETFX_CORE	Defined when building scripts against .NET Core class libraries on .NET.
NET_2_0	Defined when building scripts against .NET 2.0 API compatibility level on Mono and IL2CPP.
NET_2_0_SUBSET	Defined when building scripts against .NET 2.0 Subset API compatibility level on Mono and IL2CPP.
NET_4_6	Defined when building scripts against .NET 4.x API compatibility level on Mono and IL2CPP.
NET_STANDARD_2_0	Defined when building scripts against .NET Standard 2.0 API compatibility level on Mono and IL2CPP.
ENABLE_WINMD_SUPPORT	Defined when Windows Runtime support is enabled on IL2CPP and .NET. See Windows Runtime Support for more details.

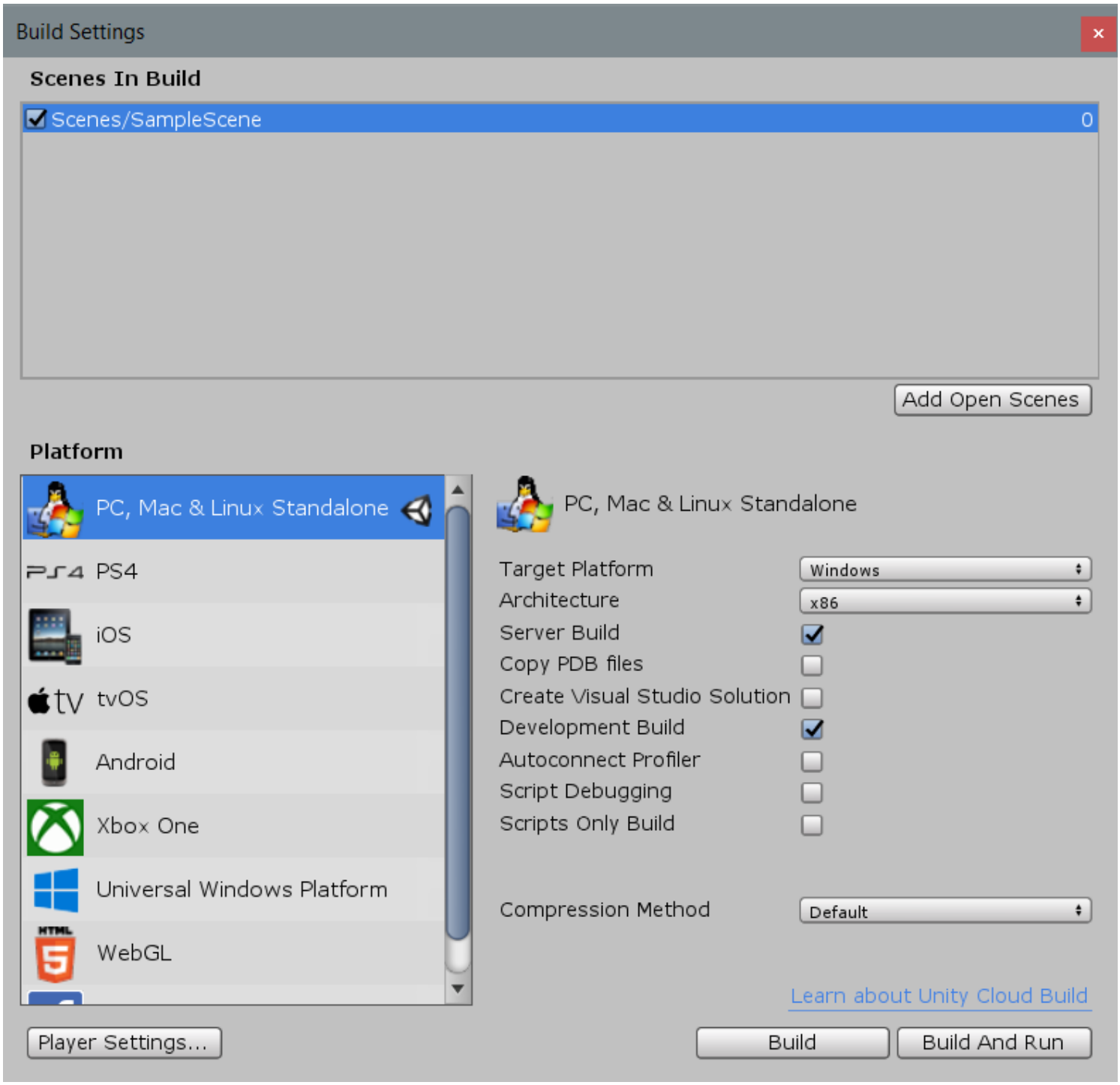
You use the DEVELOPMENT_BUILD #define to identify whether your script is running in a player which was built with the "**Development Build**" option enabled.

You can also compile code selectively depending on the scripting back-end.

Testing precompiled code

Below is an example of how to use the precompiled code. It prints a message that depends on the platform you have selected for your target build.

First of all, select the platform you want to test your code against by going to **File > Build Settings**. This displays the **Build Settings** window; select your target platform from here.



Build Settings window with PC, Mac & Linux selected as the target platforms

Select the platform you want to test your precompiled code against and click **Switch Platform** to tell Unity which platform you are targeting.

Create a script and copy/paste the following code:

```
// JS
function Awake() {
  #if UNITY_EDITOR
    Debug.Log("Unity Editor");
  #endif
```

```
  #if UNITY_IPHONE
    Debug.Log("Iphone");
  #endif

  #if UNITY_STANDALONE_OSX
    Debug.Log("Stand Alone OSX");
  #endif

  #if UNITY_STANDALONE_WIN
    Debug.Log("Stand Alone Windows");
  #endif
}


// C#
using UnityEngine;
using System.Collections;

public class PlatformDefines : MonoBehaviour {
  void Start () {

    #if UNITY_EDITOR
      Debug.Log("Unity Editor");
    #endif

    #if UNITY_IOS
      Debug.Log("Iphone");
    #endif

    #if UNITY_STANDALONE_OSX
    Debug.Log("Stand Alone OSX");
    #endif

    #if UNITY_STANDALONE_WIN
      Debug.Log("Stand Alone Windows");
    #endif

  }
}
```

To test the code, click **Play Mode**. Confirm that the code works by checking for the relevant message in the Unity console, depending on which platform you selected - for example, if you choose **iOS**, the message “Iphone” is set to appear in the console.

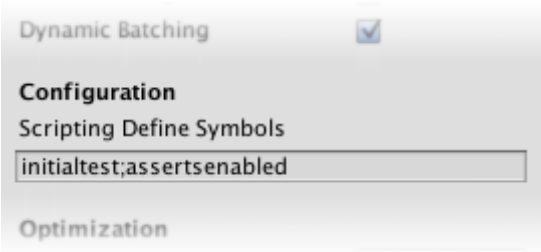
In C# you can use a `CONDITIONAL` attribute which is a more clean, less error-prone way of stripping out functions. See ConditionalAttribute Class for more information. Note that common Unity callbacks (ex. Start(), Update(), LateUpdate(), FixedUpdate(), Awake()) are not affected by this attribute because they are called directly from the engine and, for performance reasons, it does not take them into account.

In addition to the basic `#if` compiler directive, you can also use a multiway test in C#:

```
#if UNITY_EDITOR
    Debug.Log("Unity Editor");

#elif UNITY_IOS
    Debug.Log("Unity iPhone");

#else
    Debug.Log("Any other platform");

#endif
```

Platform custom #defines

It is also possible to add to the built-in selection of #define directives by supplying your own. Open the **Other Settings** panel of the Player Settings and navigate to the **Scripting Define Symbols** text box.



Enter the names of the symbols you want to define for that particular platform, separated by semicolons. These symbols can then be used as the conditions for `#if` directives, just like the built-in ones.

Global custom #defines

You can define your own preprocessor directives to control which code gets included when compiling. To do this you must add a text file with the extra directives to the **Assets** folder. The name of the file depends on the language you are using. The extension is **.rsp**:

C# (player and editor scripts) <Project Path>/Assets/mcs.rsp

As an example, if you include the single line `-define:UNITY_DEBUG` in your **mcs.rsp** file, the #define directive `UNITY_DEBUG` exists as a global #define for C# scripts, except for Editor scripts.

Every time you make changes to **.rsp** files, you need to recompile in order for them to be effective. You can do this by updating or reimporting a single script (.js or .cs) file.

NOTE

If you want to modify only global #define directives, use **Scripting Define Symbols** in Player Settings, as this covers all the compilers. If you choose the **.rsp** files instead, you need to provide one file for every compiler Unity uses, and you don't know when one or another compiler is used.

The use of **.rsp** files is described in the 'Help' section of the **mcs** application which is included in the Editor installation folder. You can get more information by running `mcs -help`.

Note that the **.rsp** file needs to match the compiler being invoked. For example:

when targeting any players or the editor, **mcs** is used with `mcs.rsp`, and
when targeting MS compiler, **csc** is used with `csc.rsp`, **etc**.

2018–03–16 Page amended with no editorial review

Removed Samsung TV support.

Special folders and script compilation order

Leave feedback

Unity reserves some project folder names to indicate that the contents have a special purpose. Some of these folders have an effect on the order of script compilation. These folder names are:

Assets
Editor
Editor default resources
Gizmos
Plugins
Resources
Standard Assets
StreamingAssets

See Special folder names for information on what these folders are used for.

There are four separate phases of script compilation. The phase where a script is compiled is determined by its parent folder.

This is significant in cases where a script must refer to classes defined in other **scripts**. The basic rule is that anything that is compiled in a phase *after* the current one cannot be referenced. Anything that is compiled in the current phase or an earlier phase is fully available.

The phases of compilation are as follows:

Phase 1: Runtime scripts in folders called **Standard Assets**, **Pro Standard Assets** and **Plugins**.
Phase 2: Editor scripts in folders called **Editor** that are anywhere inside top-level folders called **Standard Assets**, **Pro Standard Assets** and **Plugins**.
Phase 3: All other scripts that are not inside a folder called **Editor**.
Phase 4: All remaining scripts (those that are inside a folder called **Editor**).
Note: Standard Assets work only in the **Assets** root folder.

Script compilation and assembly definition files

Leave feedback

About

Unity automatically defines how **scripts** compile to managed assemblies. Typically, compilation times in the Unity Editor for iterative script changes increase as you add more **scripts** to the Project.

Use an assembly definition file to define your own managed assemblies based upon scripts inside a folder. To do this, separate Project scripts into multiple assemblies with well-defined dependencies in order to ensure that only required assemblies are rebuilt when making changes in a script. This reduces compilation time. Think of each managed assembly as a single library within the Unity Project.

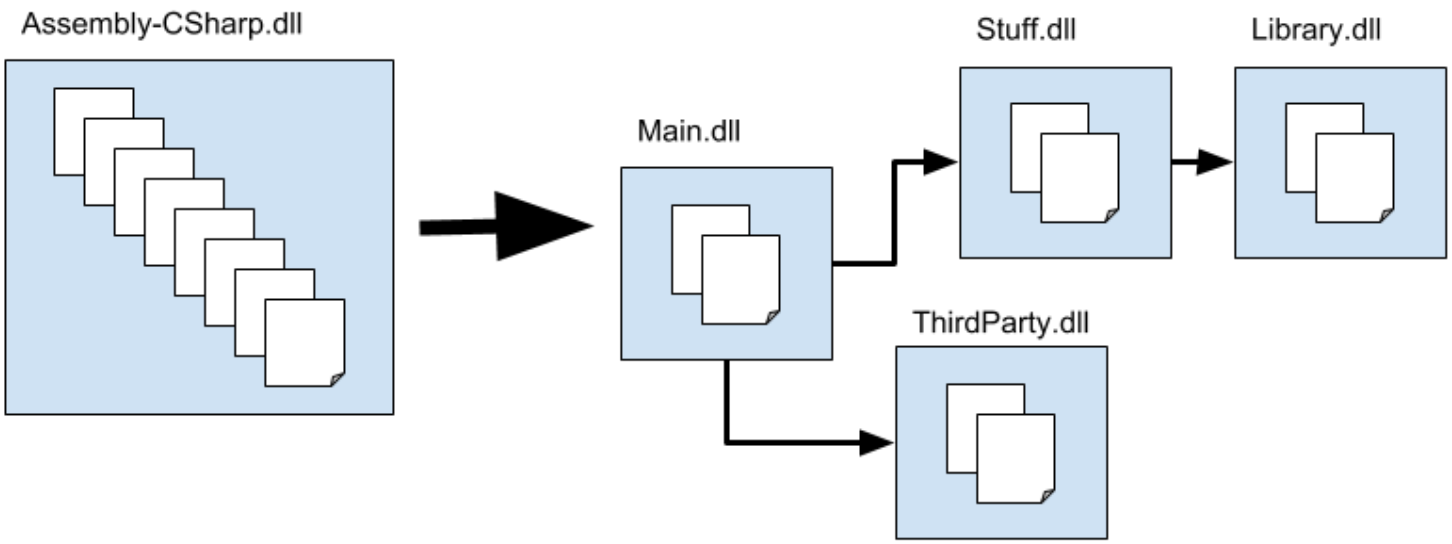


Figure 1 - Script compilation

Figure 1 above illustrates how to split the Project scripts into several assemblies. Changing only scripts in Main.dll causes none of the other assemblies to recompile. Since Main.dll contains fewer scripts, it also compiles faster than Assembly-CSharp.dll. Similarly, script changes in only Stuff.dll causes Main.dll and Stuff.dll to recompile.

How to use assembly definition files

Assembly definition files are Asset files that you create by going to **Assets** > **Create** > **Assembly Definition**. They have the extension .asmdef.

Add an assembly definition file to a folder in a Unity Project to compile all the scripts in the folder into an assembly. Set the name of the assembly in the **Inspector**.

Note: The name of the folder in which the assembly definition file resides and the filename of the assembly definition file have no effect on the name of the assembly.

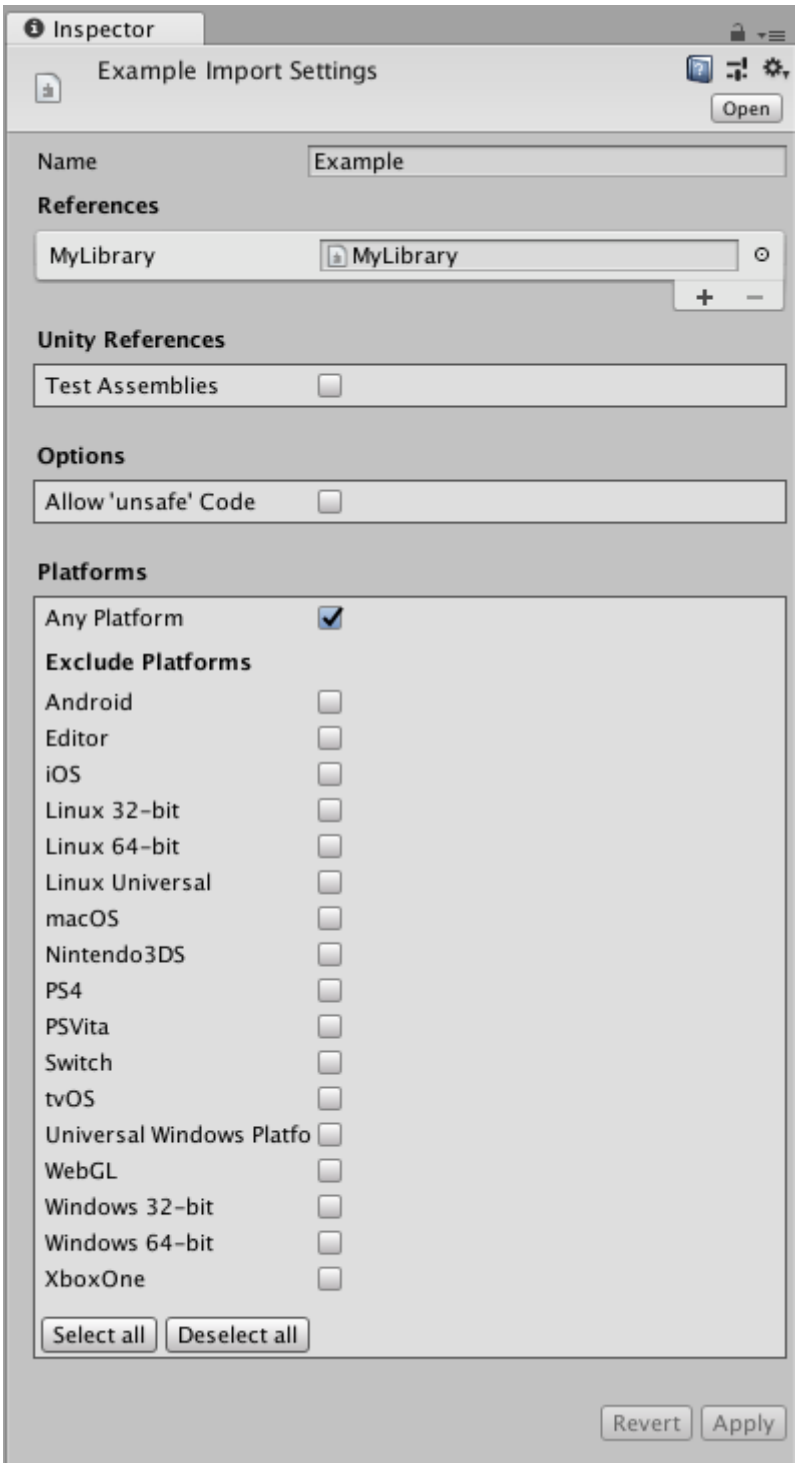


Figure 2 - Example Import Settings

Add references to other assembly definition files in the Project using the Inspector too. To view the Inspector, click your Assembly Definition file and it should appear. To add a reference, click the **+** icon under the **References** section and choose your file.

Unity uses the references to compile the assemblies and also defines the dependencies between the assemblies.

To mark the assembly for testing, enable Test Assemblies in the Inspector. This adds references to unit.framework.dll and UnityEngine.TestRunner.dll in the Assembly Definition file.

When you mark an assembly for testing, makes sure that:

Predefined assemblies (Assembly-CSharp.dll etc.) do not automatically reference Assembly Definition Files flagged for testing.

The assembly is not included in a normal build. To include the assemblies in a player build, use BuildOptions.IncludeTestAssemblies in your building script. Note that this only includes the assemblies in your build and does not execute any tests.

Note: If you use the *unsafe* keyword in a script inside an assembly, you must enable the **Allow 'unsafe' Code** option in the Inspector. This will pass the /unsafe option to the C# compiler when compiling the assembly.

You set the platform compatibility for the assembly definition files in the Inspector. You have the option to exclude or include specific platforms.

Multiple assembly definition files inside a folder hierarchy

Having multiple assembly definition files (extension: .asmdef) inside a folder hierarchy causes each script to be added to the assembly definition file with the shortest path distance.

Example:

If you have an *Assets/ExampleFolder/MyLibrary.asmdef* file and an *Assets/ExampleFolder/ExampleFolder2/Utility.asmdef* file, then:

Any scripts inside the **Assets** > **ExampleFolder** > **ExampleFolder2** folder will be compiled into the Assets/ExampleFolder/ExampleFolder2/Utility.asmdef defined assembly.

Any files in the **Assets** > **ExampleFolder** folder that are not inside the **Assets** > **ExampleFolder** > **ExampleFolder2** folder will be compiled into the Assets/ExampleFolder/MyLibrary.asmdef defined assembly.

Assembly definition files are not build system files

Note: The assembly definition files are not assembly build files. They do not support conditional build rules typically found in build systems.

This is also the reason why the assembly definition files do not support setting of preprocessor directives (defines), as those are static at all times.

Backwards compatibility and implicit dependencies

Assembly definition files are backwards compatible with the existing Predefined Compilation System in Unity. This means that the predefined assemblies always depend on every assembly definition file's assemblies. This is similar to how all scripts are dependent on all precompiled assemblies (plugins / .dlls) compatible with the active build target in Unity.

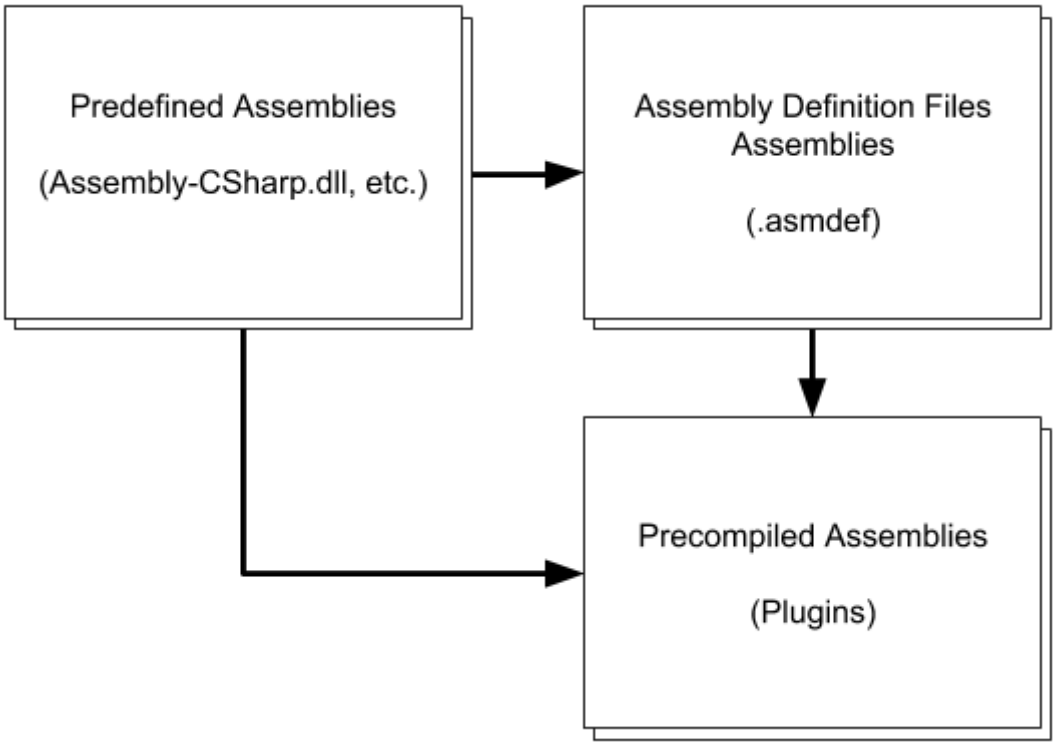


Figure 3 - Assembly dependencies

The diagram in **Figure 3** illustrates the dependencies between predefined assemblies, assembly definition files assemblies and precompiled assemblies.

Unity gives priority to the assembly definition files over the Predefined Compilation System. This means that having any of the special folder names from the predefined compilation inside a assembly definition file folder does not have any effect on the compilation. Unity treats these as regular folders without any special meaning.

It is highly recommended that you use assembly definition files for all the scripts in the Project, or not at all. Otherwise, the scripts that are not using assembly definition files always recompile every time an assembly definition file recompiles. This reduces the benefit of using assembly definition files.

API

In the UnityEditor.Compilation namespace there is a static CompilationPipeline class that you use to retrieve information about assembly definition files and all assemblies built by Unity.

File format

Assembly definition files are JSON files. They have the following fields:

Field	Type
name	string
references(optional)	string array
includePlatforms (optional)	string array
excludePlatforms (optional)	string array

Do not use the **includePlatforms** and **excludePlatforms** fields in the same assembly definition file.

Retrieve the platform names by using `CompilationPipeline.GetAssemblyDefinitionPlatforms`.

Examples

MyLibrary.asmdef

```
{
    "name": "MyLibrary",
    "references": [ "Utility" ],
    "includePlatforms": ["Android", "iOS"]
}
```

MyLibrary2.asmdef

```
{
    "name": "MyLibrary2",
    "references": [ "Utility" ],
     "excludePlatforms": ["WebGL"]
}
```

2018–03–07 Page published with limited editorial review

New feature in 2017.3

Custom Script Assemblies updated in 2018.1

.NET profile support

Leave feedback

Unity supports a number of .NET profiles. Each profile provides a different API surface for C# code which interacts with the .NET class libraries. You can change the .NET Profile in the Player Settings (go to **Edit** > **Project Settings** > **Player**) using the **Api Compatibility Level** option in the **Other Settings** section.

Legacy scripting runtime

The legacy scripting runtime supports two different profiles: .NET 2.0 Subset and .NET 2.0. Both of these are closely aligned with the .NET 2.0 profile from Microsoft. The .NET 2.0 Subset profile is smaller than the .NET 4.x profile, and it allows access to the class library APIs that most Unity projects use. It is the ideal choice for size-constrained platforms, such as mobile, and it provides a set of portable APIs for multiplatform support. By default, most Unity projects should use the .NET Standard 2.0 profile.

Stable scripting runtime

The stable scripting runtime supports two different profiles: .NET Standard 2.0 and .NET 4.x. The name of the .NET Standard 2.0 profile can be a bit misleading because it is not related to the .NET 2.0 and .NET 2.0 Subset profile from the legacy scripting runtime. Instead, Unity's support for the .NET Standard 2.0 profile matches the profile of the same name published by the .NET Foundation. The .NET 4.x profile in Unity matches the .NET 4 series (.NET 4.5, .NET 4.6, .NET 4.7, and so on) of profiles from the .NET Framework.

Only use the .NET 4.x profile for compatibility with external libraries, or when you require functionality that is not available in .NET Standard 2.0.

Cross-platform compatibility

Unity aims to support the vast majority of the APIs in the .NET Standard 2.0 profile on all platforms. While not all platforms fully support the .NET Standard, libraries which aim for cross-platform compatibility should target the .NET Standard 2.0 profile. The .NET 4.x profile includes a much larger API surface, including parts which may work on few or no platforms.

Managed plugins

Managed code **plugins** compiled outside of Unity can work with either the .NET Standard 2.0 profile or the .NET 4.x profile in Unity. The following table describes configurations Unity supports:

	API Compatibility Level:	
	.NET Standard 2.0	**.NET 4.x**
Managed plugin compiled against:		
.NET Standard	Supported	Supported
.NET Framework	Limited	Supported
.NET Core	Not Supported	Not Supported

Note:

Managed plugins compiled against any version of the .NET Standard work with Unity.
Limited support indicates that Unity supports the configuration if all APIs used from the .NET Framework are present in the .NET Standard 2.0 profile. However, the .NET Framework API is a superset of the .NET Standard 2.0 profile, so some APIs are not available.

Transport Layer Security (TLS) 1.2

From 2018.2, Unity provides full TLS 1.2 support on all platforms except **WebGL**. It does this via the UnityWebRequest API and all .NET 4.x APIs.

Certificate verification is done automatically via the platform specific certificate store if available. Where access to the certificate store is not possible, Unity uses an embedded root certificate store.

TLS support for .NET 3.5 and below varies per platform and there are no guarantees on which features are supported.

2018–03–15 Page amended with editorial review

.NET profile support added in 2018.1

Referencing additional class library assemblies

Leave feedback

If a Unity Project needs access to a part of the .NET class library API that is not compiled by default, the Project can inform the C# compiler in Unity. The behavior depends which .NET profile the Project uses.

.NET Standard 2.0 profile

If your Projects use the .NET Standard 2.0 **Api Compatibility Level**, you shouldn't need to take any additional steps to use part of the .NET class library API. If part of the API seems to be missing, it might not be included with .NET Standard 2.0. The Project may need to use the .NET 4.x **Api Compatibility Level** instead.

.NET 4.x profile

By default, Unity references the following assemblies when using the .NET 4.x **Api Compatibility Level**:

mscorlib.dll
System.dll
System.Core.dll
System.Runtime.Serialization.dll
System.Xml.dll
System.Xml.Linq.dll

You should reference any other class library assemblies using an *mcs.rsp* file. You can add this file to the **Assets** directory of a Unity Project, and use it to pass additional command line arguments to the C# compiler. For example, if a Project uses the `HttpClient` class, which is defined in the *System.Net.Http.dll* assembly, the C# compiler might produce this initial error message:

```
The type `HttpClient` is defined in an assembly that is not referenced. You must add a referenc
```

You can resolve this error by adding the following **mcs**.rsp file to the Project:

```
-r:System.Net.Http.dll
```

You should reference class library assemblies as described in the example above. Don't copy them into the Project directory.

Switching between profiles

Exercise caution when using an *mcs.rsp* file to reference class library assemblies. If you change the **Api Compatibility Level** from .NET 4.x to .NET Standard 2.0, and an *mcs.rsp* like the one in the example above exists in the Project, then C# compilation fails. The *System.Net.Http.dll* assembly does not exist in the .NET Standard 2.0 profile, so the C# compiler is unable to locate it.

The *mcs.rsp* file can have parts that are specific to the current .NET profile. If you make changes to the profile, you need to modify the *mcs.rsp* file.

2018–03–15 Page amended with editorial review